

- 从信息检索的核心概念入手，介绍Lucene与分布式搜索服务器Elasticsearch的相关知识
- 从原理到实践，涵盖Elasticsearch 5.4 开发所需要的各种技术
- 以实例为导向，并提供可执行程序与详尽的代码注解，有效降低学习门槛

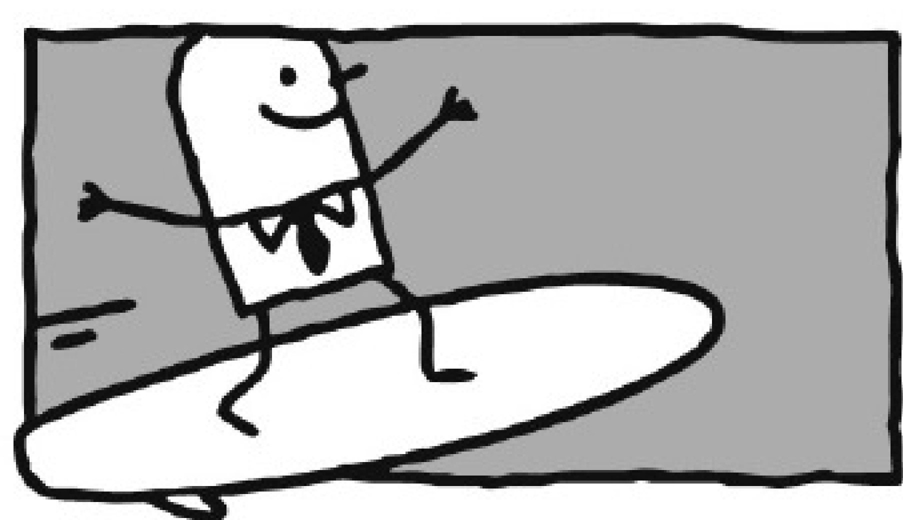
从Lucene到Elasticsearch全文检索实战

姚攀 编著

大数据时代的信息检索技术



清华大学出版社



从 Lucene 到 Elasticsearch 全文检索实战

姚 攀 编著

清華大學出版社
北 京

内 容 简 介

本书循序渐进介绍了信息检索、布尔检索、向量空间模型、tf-idf、BM25 排序算法、Lucene 架构、Lucene 创建索引、Lucene 查询、Lucene 项目实战、Elasticsearch 安装与配置、Elasticsearch 插件安装、REST API 数据操作、映射与模板、索引别名、Elasticsearch 基本和高级搜索、Elasticsearch 同步数据库、Elasticsearch 集群管理、项目实战等内容。阅读本书，读者能够掌握信息检索的核心概念，应用 Lucene 库处理全文检索业务，掌握 Elasticsearch 分布式搜索引擎的使用方法与技巧。

本书基于 Lucene 6.0 和 Elasticsearch 5.4.0 进行讲解，技术先进，示例丰富，适合想学习信息检索技术的初学者和相关专业的大学生、研究生学习，也很适合大数据及云计算平台构建人员以及有一定基础的 IT 开发人员使用。

图书在版编目（CIP）数据

从 Lucene 到 Elasticsearch：全文检索实战/姚攀编著. —北京：清华大学出版社，2017（2018.6 重印）
ISBN 978-7-302-48306-9

Ⅰ. ①从… Ⅱ. ①姚… Ⅲ. ①全文检索 Ⅳ. ①G254.923

中国版本图书馆 CIP 数据核字（2017）第 215137 号

责任编辑：王金柱
封面设计：王　翔
责任校对：闫秀华
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　　张：**20.75　　**字　　数：**531 千字
版　　次：2017 年 12 月第 1 版　　**印　　次：**2018 年 6 月第 2 次印刷
印　　数：3001~4500
定　　价：79.00 元

产品编号：072144-02

前　　言

我们正处在一个大数据时代，大数据并不仅仅是指海量数据，而更多的是指这些数据都是非结构化的、无法用传统的方法进行处理的数据。相信很多人听说过目前在云计算和大数据领域里如日中天的 Hadoop，Hadoop 的发起人之一是大名鼎鼎的 Doug Cutting。早在 Hadoop 诞生之前，Doug Cutting 已经用 Java 实现了第一个提供全文文本搜索的开源函数库 Lucene。Lucene 自 2000 年发布第一个开源版本以来，在开源社区引起了很大的反响，为广大开发者提供了研发全文检索系统的利器。Lucene 作为 Apache 的顶级项目，有大量研发人员贡献源码，经过十几年的发展，目前 Lucene 已经十分成熟，可以说 Lucene 是当今最先进、最高效的全功能开源搜索引擎工具包。但 Lucene 只是一个全文检索类库，Elasticsearch 是一个建立在 Lucene 基础上的实时的分布式搜索引擎，2010 年由 Shay Bano 发布。相比于 Lucene，Elasticsearch 功能更加强大，使用更加方便。

站在巨人的肩膀上，入门搜索技术并不困难，本书为入门 Lucene、Elasticsearch 而生。本书首先介绍信息检索领域中的一些基本理论，也就是 Lucene 的数学模型，之后介绍如何使用 Lucene 库构建全文检索系统，最后介绍 Elasticsearch。本书按照从数学模型到入门基础再到项目实战的思路来编写，数学模型让读者知其然也知其所以然，入门基础是理论到实际应用的必经之路，项目实战则是为了学以致用。书中的每一部分都力图简明扼要，使用大量实例和代码，为读者能够快速掌握全文检索技术扫除障碍。将全文检索领域中的一些知识和项目经验分享给大家，是笔者写作本书的初衷。

本书结构

本书从逻辑上可划分为三部分。

第一部分（第 1 章），主要介绍信息过载、信息检索、倒排索引、布尔模型、tf-idf、向量空间模型、概率检索模型等信息检索领域的基础知识。

第二部分（第 2 和 3 章），介绍如何使用 Lucene 开发全文检索系统。

第 2 章主要介绍 Lucene 的基础知识，内容包括 Lucene 的特点、Lucene 架构、Luke 的使用、IK 分词器配置、扩展词库和远程词库的配置、Lucene 的多种分词器、索引的构建方法、检索文档以及实现检索关键词高亮的方法。

第 3 章是 Lucene 项目实战部分，介绍如何使用 Lucene 构建一个文件检索系统，内容包括项目的整体设计、使用 Tika 做信息抽取、索引的构建、用户查询界面的设计与实现、用户查询处理、搜索结果展示等内容。

第三部分（第 4~11 章），主要介绍 Elasticsearch 分布式搜索引擎的相关技术。

第 4 章是 Elasticsearch 简介，内容包括 Elasticsearch 与 Lucene 的关系、Elasticsearch 的整体架构、核心概念、在企业中的应用案例、流行度趋势、Elasticsearch 的安装、中文分词配置以及相关插件的安装与使用。

第 5 章是 Elasticsearch 集群入门，主要内容包括索引管理、文档管理和映射详解。

第 6 章介绍 Elasticsearch 的搜索功能，主要内容包括搜索机制的解读、全文查询、词项查询、复合查询、嵌套查询、位置查询、特殊查询、搜索高亮和排序。

第 7 章介绍 Elasticsearch 的聚合分析功能。

第 8 章介绍如何使用 Elasticsearch Java API 做二次开发。

第 9 章介绍 Elasticsearch 集群管理的相关知识点，包括脑裂问题、集群规划、索引规划、分布式集群的搭建方法以及如何查看集群的监控信息。

第 10 章是 Elasticsearch 整合 MySQL 项目实战部分，通过实现对 MySQL 中的数据进行全文检索这一需求，贯穿了 MySQL、JDBC、Elasticsearch Java API 以及 Java Web 的相关知识，使读者了解在实际的项目开发中使用 Elasticsearch 做全文搜索的方法。

第 11 章介绍 Elasticsearch 和 Hadoop 大数据平台交互的方法。

学习本书的预备知识

Java 基础

首先要配置好 Java 开发环境。不论是学习 Lucene 还是 Elasticsearch 都需要安装好 Java 环境，Elasticsearch 的运行要求 JDK 版本最低为 1.7，建议使用 JDK 1.8 及以上版本。鉴于 Java 的跨平台特性，对操作系统没有要求，在 Windows、Linux、Mac OS X 系统上都可以运行 Elasticsearch。除此之外，读者需要掌握 Java 基础知识。

Java Web 开发技术

在项目实战中需要用到 Java Web 的相关技术，建议读者在阅读本书之前掌握 HTML、CSS、JSP 等基础知识，掌握 Java Web 项目的部署和运行。

本书使用的软件版本

本书基于 Lucene 6.0 和 Elasticsearch 5.4.0 进行讲解，集成开发环境为 Eclipse 4.6.1。

读者对象

在校学生

如果你是正在大学校园里修读计算机科学相关专业的大学生，也许你正在选修程序设计语言，课程结束了你发现自己只能写出命令行下黑白屏显示的小程序，你也许很期待学到更多的技术做出实际的项目，那么本书就是为你准备的。书中的项目使用的是 Java 语言，除了 Lucene 和 Elasticsearch 的使用之外，还穿插了 Java SE、Java Web 的相关技术。

Java 程序开发者

如果你是已经参加工作的 Java 程序开发者，想要掌握全文检索相关技术却不知道从哪里入手，需要处理企业中的全文检索业务却没有思路，你也许听说过 Lucene 或 Elasticsearch，但是不知道怎样快速入门，那么本书可以作为入门全文检索、学习 Lucene 和 Elasticsearch 开发技术的参考书。

搜索引擎研发人员

如果你是搜索引擎研发者，本书中的实际案例和相关知识点可以作为参考资料，比如信息检索模型理论基础、文档信息抽取、Lucene 应用案例、Elasticsearch Java API、Elasticsearch 集群管理等。希望能以本书为媒介和大家共同探讨和交流。

源代码下载

源代码下载地址：https://pan.baidu.com/s/1IHN6i8_6RRcua3cf5za5Ww（注意区分数字和英文字母的大小写）

勘误与交流

限于笔者水平和写作时间有限，不可避免地会有些疏漏之处，欢迎大家通过电子邮件等方式批评指正。

笔者的邮箱：ucasyp@163.com

笔者的博客：blog.csdn.net/napoay

QQ 交流群：614836679

致谢

本书能够顺利出版要感谢很多单位和个人。首先要感谢笔者的家人，他们对笔者学业的支持和生活的照顾使笔者没有后顾之忧，全身心投入到本书的写作当中。

感谢北京博瑞开源有限公司，公司给笔者提供了宝贵的实习机会，本书的很多知识点都来源于实际项目，是在解决实际问题过程中的经验总结，感谢公司董事长李小翔先生、架构师黄超对笔者的指导和帮助。

感谢马玉鹏老师、郎睿师兄、张港红博士、CSDN 博主周程（blog.csdn.net/fxsdbt520）、秦雪箭、宗鹏、陆风光在本书写作过程中的帮助和支持。

感谢启明星辰的张铭清女士在第二次印刷前对本书的指正和建议。

感谢清华大学出版社给了笔者一次和大家分享技术、交流学习的机会，感谢王金柱编辑在本书出版过程的辛勤付出。

中国科学院大学　姚　攀

2018 年 5 月 4 日

目　录

第 1 章

信息检索模型

本章学习要点：

- ✱ 信息过载简介
- ✱ 信息检索定义
- ✱ 分词算法简介
- ✱ 倒排索引介绍
- ✱ 布尔检索模型
- ✱ tf-idf 权重计算
- ✱ 向量空间模型
- ✱ 概率检索模型

1.1　信息检索概述

1.1.1　信息过载

互联网的飞速发展使人类进入了信息大爆炸的时代，根据相关统计数据显示，目前全球网民数量已经达到 32 亿人，互联网上的数据也呈指数级增长。图 1-1 是国外初创公司 Domo 发布的一张信息图，该图展示了各大网站在 60 秒内产生的巨大数据量。

根据 Domo 的数据显示，在一分钟内 YouTube 用户会上传 400 个小时的新视频，Netflix 用户每分钟则观看 86 805 个小时的视频。与此同时，苹果用户每分钟下载 51 000 个应用，亚马逊每分钟交易额达 222 283 美元，Google 在一分钟内翻译了 69 500 000 个单词，SIRI 一分钟内回答了 99 206 个问题，The Weather Channel（注：一款天气预报软件）每分钟接收 13 888 889 次天气查询请求。在社交网站方面，Facebook 用户每分钟分享 216 302 张照片，Dropbox 用户每分钟上传 833 333 个新文件，Tinder 用户每分钟发布 9678 条表情符号推文，而 Snapchat 用户每分钟会发布 284 722 张照片。

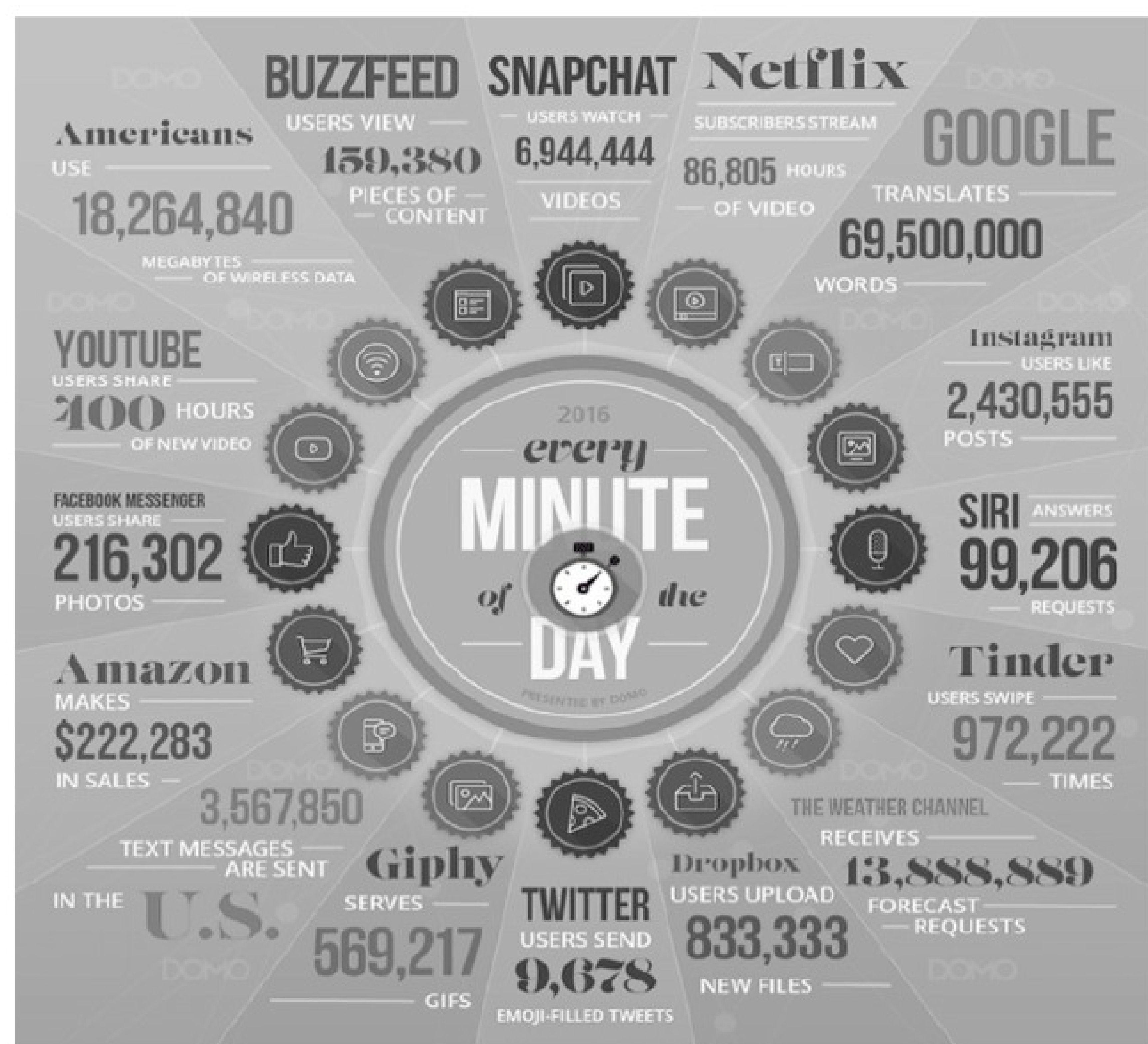


图 1-1　互联网上的一分钟

就在你看完上面这段内容的时间里，所有的一切都可能发生了改变。我们处在一个大数据时代，也是一个信息过载（Information Overload）的时代。大数据时代的特点可以用 4 个 V 来概括：

- Volume
  数据量大，全球每年产生的数据总量已经达到 ZB（1ZB=2^{40}GB）级别。
- Variety
  数据种类繁多，如文本、图片、视频、地理信息、各种传感器信息等。
- Velocity
  数据流动速度快，对数据处理的时效性要求高。
- Value
  大数据蕴含着巨大的价值，可以帮助人们解决数据量不足时所不能解决的问题。

信息过载是指社会信息超过了个人或系统所能接受、处理或有效利用的范围，并导致故障的状况。信息过载主要有以下 3 个特点：

（1）受传者对信息反映的速度远远低于信息传播的速度。

（2）大众媒介中的信息量大大高于受众所能消费、承受或需要的信息量。

（3）大量无关的、没用的、冗余的信息严重干扰了受众对相关有用信息的准确分析和正确选择。

信息过载是信息时代信息极大丰富的负面影响之一。

1.1.2　信息检索定义

信息资源总量呈爆炸式增长，在信息的海洋里获取想要的信息变得更加困难。为了解决

信息过载的问题，无数科学家和工程师提出了很多天才的解决方案，其中最具代表性的是分类目录和搜索引擎。

分类目录是将网站信息系统地分类整理，提供一个按类别编排的网站目录，在每类中排列着属于这一类别的网站站名、网址链接、内容提要以及子分类目录，可以在分类目录中逐级浏览寻找相关的网站，分类目录中往往还提供交叉索引，从而可以方便地在相关的目录之间跳转和浏览。互联网早期的门户网站，比如雅虎、搜狐、新浪等，都是将不同来源的信息以一种整齐划一的形式整理、储存并呈现给用户，用户根据信息来源、信息类型、关键字等方式筛选网站内容。

搜索引擎是指自动从因特网搜集信息，经过一定整理以后，提供给用户进行查询的系统。国外具有代表性的搜索引擎有 Google、Bing、Yahoo 等，国内具有代表性的搜索引擎有百度搜索、搜狗搜索、360 搜索等。

我们常用的搜索引擎是 Web 搜索，是信息检索的一个分支，学术上的信息检索（Information Retrieval，简称 IR）的定义为：信息检索是从大规模非结构化数据（通常是文本）的集合（通常保存在计算机上）中找出满足用户信息需求的资料（通常是文档）的过程。

1.1.3　信息检索常用术语

信息检索领域有一些常用的术语，深刻理解这些术语对入门信息检索非常有必要，简介如下。

- 用户需求（User Need，简称 UN）
  用户需要获得的信息。严格地说，UN 只存在于用户的内心，但是通常用文本来描述，如查找与“2014 世界杯”相关的新闻，有时也称为主题（Topic）。
- 查询（Query）
  UN 提交给检索系统时称为查询（Query），如“iPhone7 价格”。对同一个 UN，不同人不同时候可以构造出不同的 Query，上述需求也可表示成“苹果 7 价格”。Query 在 IR 系统中往往还有内部表示。
- 文档（Document）
  文档是信息检索的对象，文档不仅仅可以是文本，也可以是图像、视频、语音等多媒体文档。
- 文档集（Crops）
  由若干文档构成的集合称为文档集合，文档集有时也称为语料库。海量的互联网网页、文件系统中的文本文件、大量的电子邮件，都是文档集合的具体例子。
- 文档编号（Document ID）
  文档 ID 是给文档集中的每个文档赋予的唯一标识符，通过文档 ID 来区分不同的文档，这样能够方便搜索引擎的内部处理。缩写为 docID。
- 词条化（tokenization）
  词条化是将给定的字符序列拆分成一系列子序列的过程，拆分的每个子序列称为一个词条。词条化的过程中有可能会去除标点符号等特殊字符。下面是一个词条化的具体例子。

  输入：Whatever happens tomorrow,we have had today.

  输出：| whatever | happens | tomorrow | we | have | had | today |

- 词项（Term）
  词项是经过语言学预处理之后归一化的词条。词项是索引的最小单位，一般情况下可以把词项当作词，但词项不一定就是词。对于上面的句子，产生词项如下：

whatever	happen	tomorrow	we	have	had	today

- 词项-文档关联矩阵（Incidence matrix）
  词项-文档关联矩阵是表示词项和文档之间所具有的一种包含关系的概念模型，表 1-1 展示了其含义。表中的每列代表一个文档，每行代表一个词项，打对勾的位置代表包含关系。

表1-1　词项－文档关联矩阵

	doc1	doc2	doc3	doc4	doc5	doc6
term1	√		√			√
term2		√			√	
term3		√		√		
term4	√		√		√	
term5	√			√		√
term6		√	√		√	

从纵向即文档这个维度来看，每列代表一个文档包含的词项信息，比如 doc1 包含了 term1、term4 和 term5，而不包含 term2、term3、term6。从横向即词项这个维度来看，每行代表该词项在文档中的分布信息，比如对于 term1 来说，doc1、doc3、doc6 中出现过 term1，而其他文档不包含 term1。矩阵中其他的行列也可作此种解读。

- 词项频率（Term frequency）
  同一个单词在某个文档中出现的频率。比如，单“apple”在某文档中出现了 3 次，那么该单词在该文档中的词项频率就是 3。
- 文档频率（Document frequency）
  出现某词项的文档的数目。比如，单词“China”只出现在文档集合中的文档 1 和文档 5，那么该单词的文档频率就是 2。
- 倒排记录表（Postings lists）
  倒排记录表用于记录出现过某个单词的所有文档的文档列表以及单词在该文档中出现的位置信息，每条记录称为一个倒排项。通过倒排列表即可获知哪些文档包含哪些单词。
- 倒排文件（Inverted file）
  倒排记录表在磁盘中的物理存储文件称为倒排文件。

1.1.4　信息检索系统

一个完整的信息检索系统的基本架构如图 1-2 所示。信息检索系统可以分为信息采集、信息整理和用户查询 3 部分。

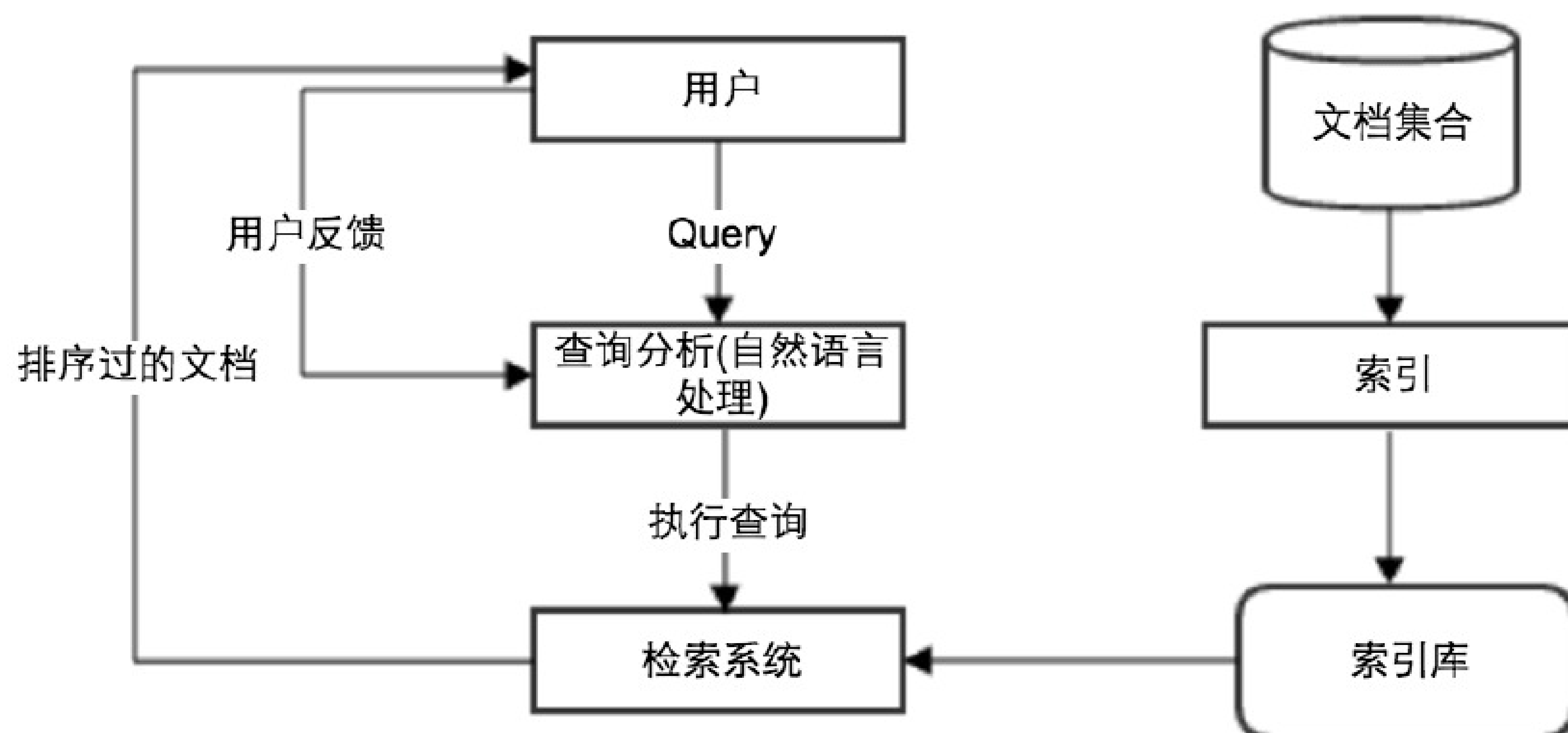


图 1-2　IR 系统基本架构图

1. 信息采集

信息采集基本都是通过网络爬虫（Spider）自动完成的。网络爬虫是一种按照一定的规则，自动地抓取万维网信息的程序或者脚本。互联网上的网页数以亿计，遍布在全球的各个服务器上，通过爬虫可以将网页下载下来进行进一步的分析和挖掘，经过格式处理之后提取网页信息为构建索引做准备。

2. 整理信息

信息检索系统整理信息的过程称为索引构建。信息检索系统不仅要保存搜集起来的信息，还要将它们按照一定的规则进行编排，这样就不用重新翻查它所有保存的信息就能迅速找到所要的资料。如果信息是不按任何规则地随意堆放在系统中，那么它每次找资料都得把整个资料库完全翻查一遍，如此一来再快的计算机系统也没有用。

3. 接受查询

用户向信息检索系统发出查询请求，信息检索系统接受查询并向用户返回检索到的文档。信息检索系统（尤其是商用搜索引擎）每时每刻都要接到来自大量用户的几乎是同时发出的查询，它按照每个用户的要求检查自己的索引，在极短时间内找到用户需要的文档，并返回给用户。目前，搜索引擎返回主要是以网页链接的形式提供的，通过这些链接用户便能到达含有自己所需资料的网页。搜索引擎通常会在这些链接下提供一小段来自这些网页的摘要信息以帮助用户判断此网页是否含有自己需要的内容。

1.2　分词算法

1.2.1　分词算法概述

词是表达语义的最小单位。分词对搜索引擎的帮助很大，可以帮助搜索引擎程序自动识别语句的含义，从而使搜索结果的匹配度达到最高，因此分词的质量也就直接影响了搜索结果的精确度。分词在文本索引的建立过程和用户提交检索过程中都存在。利用相同的分词器，把短

语或者句子切分成相同的结果，才能保证检索过程顺利进行。中文和英文的分词原理简介如下：

1. 英文分词的原理

基本的处理流程是：输入文本、词汇分割、词汇过滤（去除停留词）、词干提取（形态还原）、大写转为小写、结果输出。

2. 中文分词原理

中文分词比较复杂，并没有英文分词那么简单。这主要是因为中文的词与词之间并不像英文中那样用空格来隔开。中文分词主要有 3 种方法：基于词典匹配的分词方法、基于语义理解的分词、基于词频统计的分词。

1.2.2　词典匹配分词法

基于字典匹配的分词方法按照一定的匹配策略将输入的字符串与机器字典词条进行匹配，这种方法是最简单的也是最容易想到的分词办法，最早由北京航空航天大学的梁南元教授提出。查字典分词实际上就是把一个句子从左向右扫描一遍，遇到字典中有的词就标识出来，遇到复合词就找到最长的词匹配，遇到不认识的字串则切分成单个词。按照匹配操作的扫描方向不同，字典匹配分词方法可以分为正向匹配、逆向匹配以及结合了两者的双向匹配算法；按照不同长度优先匹配的情况，可以分为最大（最长）匹配和最小（最短）匹配；按照是否与词性标注过程相结合，又可以分为单纯分词方法和分词与词性标注相结合的方法。几种常用的词典分词方法如下：

- 正向最大匹配（由左到右的方向）
- 逆向最大匹配（由右到左的方向）
- 最少切分（是每一句中切除的词数最小）

实际应用中上述各种方法经常组合使用，以达到最好的效果，从而衍生出了结合正向最大匹配方法和逆向最大匹配算法的双向匹配分词法。由于中文分词最大的问题是歧义处理，结合中文语言自身的特点，经常采用逆向匹配的切分算法，处理的精度高于正向匹配，产生的切分歧义现象也较少。

真正实用的分词系统，都是把词典分词作为基础手段，结合各种语言的其他特征信息来提高切分的效果和准确度。有的实用系统中将分词和词性标注结合起来，利用句法和词法分析对分词决策提高帮助，在词性标注过程中迭代处理，利用词性和语法信息对分词结果进行检验、调整。

1.2.3　语义理解分词法

基于语义理解的分词方法是模拟人脑对语言和句子的理解，达到识别词汇单元的效果。基本模式是把分词、句法、语义分析并行进行，利用句法和语义信息来处理分词的歧义。

一般结构中通常包括分词子系统、句法语义子系统、调度系统。在调度系统的协调下，分词子系统可以获得有关词、句子等的句法和语义信息，模拟人脑对句子的理解过程。基于语义理解的分词方法需要使用大量的语言知识和信息。

目前国内外对汉语语言知识的理解和处理能力还没有达到语义层面，具体到语言信息很难组织成机器可直接读取、计算的形式，因此目前基于语义理解的分词系统还处在试验阶段。

1.2.4　词频统计分词法

这种做法基于人们对中文词语的直接感觉。通常词是稳定的字的组合，因此在中文文章的上下文中，相邻的字搭配出现的频率越多，就越有可能形成一个固定的词。根据 n 元语法知识可以知道，字与字相邻同时出现的频率或概率能够较好地反映成词的可信度。实际的系统中，通过对精心准备的中文语料中相邻共现的各个字的组合的频度进行统计，计算不同字词的共现信息。根据两个字的统计信息，计算两个汉字的相邻共现概率，统计出来的信息体现了中文环境下汉字之间结合的紧密程度。当紧密程度高于某一个阈值时，便可认为此字组可能构成一个词。

基于词频统计的分词方法只需要对语料中的字组频度进行统计，不需要切分词典，因而又叫作无词典分词法或统计分词方法。这种方法经常抽出一些共现频度高但并不是词的常用字组，需要专门处理，提高精确度。实际应用的统计分词系统都使用一个基本的常用词词典，把字典分词和统计分词结合使用。基于统计的方法能很好地解决词典未收录新词的处理问题，即将中文分词中的串频统计和串匹配结合起来，既发挥匹配分词切分速度快、效率高的特点，又利用了无词典分词结合上下文识别生词、自动消除歧义的优点。

1.3　倒排索引

索引是构成搜索引擎的核心技术之一，索引在日常生活中其实也是非常常见的，比如当我们看一本书的时候，我们首先会看书的目录，通过目录可以快速定位到某一章节的页码，加快对内容的查询速度。

文档通常保存在各种数据库管理系统之中，比如 Oracle、MySQL 等。但是搜索引擎中的数据不能保存到数据库中，主要是因为数据库不能满足搜索引擎的需求，原因有二：一是搜索引擎中的数据量非常庞大，大型商业搜索引擎需要处理数以亿计的网页，面对海量数据使用关系型数据库很难管理；二是搜索引擎使用的数据操作非常简单，一般只需增删改查这几个基本功能，一般的数据库系统则支持大而全的功能，损失了速度和空间，大量用户检索则要求搜索引擎响应时间必须很快，检索效率要非常高，数据库系统在检索响应时间和检索并发度方面都不能满足需求。而数据库中的索引就是为了提高表的搜索效率而对某些字段中的值建立的目录，在搜索引擎中使用倒排索引这种数据结构来存储网页信息。

倒排索引（Inverted index），也常被称为反向索引，是一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射，它是文档检索系统中最常用的数据结构。

下面以简单通俗的例子来理解倒排索引，假设现在有两个文档 doc1 和 doc2，doc1 包含 3 个关键词：中国、美国、韩国，doc2 中包含 4 个关键词：中国、美国、德国、英国，文档和词语的包含关系（也就是正排索引），见表 1-2。

表 1-2　文档－单词对照表

文　档	词　语
doc1	中国、美国、韩国
doc2	英国、中国、美国、德国

那么词语所属的文档关系，也就是倒排索引，见表 1-3。

表 1-3　单词－文档对照表

词　语	文　档
中国	doc1、doc2
美国	doc1、doc2
韩国	doc1
英国	doc2
德国	doc2

如果想查找包含关键词“美国”的文档，那么结果就是 doc1 和 doc2。这样从文档包含单词到单词所属文档的转换，就是倒排的由来。我们在搜索引擎中输入关键词进行查询，就是一次查找哪些文档包含查询关键词的过程。

下面我们通过具体实例深入理解倒排索引，通过简单文档以小见大，体验倒排索引的构建过程。如表 1-4 所示，在互联网上找了 4 条科技新闻作为一个文档集合，我们以新闻标题作为文档内容，给每个文档设置一个连续的整数编号作为文档 ID。

表 1-4　构建倒排索引文档集合

文档 ID	文档内容
1	人工智能成为互联网大会焦点
2	谷歌推出开源人工智能系统工具
3	互联网的未来在人工智能
4	谷歌开源机器学习工具

对于文档内容，先要经过词条化处理。和英文不同的是，英文通过空格分隔单词，中文的词与词之间没有明确的分隔符号，经过分词系统进行中文分词以后把矩阵切分成一个个的词条，文档 4 会被分成“谷歌”“开源”“机器”“学习”“工具”5 个词项。“谷歌”这个词在文档 2 和文档 4 中各出现一次，文档频率为 2，倒排记录表记作 2→4，文档频率也是倒排记录表的长度。依次统计各个词项的文档频率和倒排记录表，构建倒排索引过程如表 1-5 所示。

表 1-5　倒排索引构建过程

词　项	文档频率	倒排记录表
人工	3	1→2→3
智能	3	1→2→3
成为	1	1
互联网	2	1→3

（续表）

词　项	文档频率	倒排记录表
大会	1	1
焦点	1	1
谷歌	2	2→4
推出	1	2
开源	2	2→4
系统	1	2
工具	2	2→4
的	1	3
未来	1	3
在	1	3
机器	1	4
学习	1	4

1.4　布尔检索模型

检索模型是判断文档内容与用户查询相关性的核心技术，以大规模网页搜索为例，在海量网页中与用户查询关键词相关的网页可能会有成千上万个，甚至更多。那么信息检索系统是如何判断网页和查询关键词是相关的？内部的排序模型是怎样的？

布尔检索法是指利用布尔运算符连接各个检索词，然后由计算机进行逻辑运算，找出所需信息的一种检索方法。布尔检索模型的数学基础是集合论，在该模型下每篇文档被看成是一系列词的集合。

布尔检索模型中主要有 AND、OR、NOT 三种逻辑运算，布尔逻辑运算符的作用是把检索词连接起来，构成一个逻辑检索式。

- AND（或*）：逻辑与，用来表示其所连接的两个检索项的交叉部分，即检索词的交集部分。例如检索同时含有关键词 A 和 B 的集合：A　AND　B
- OR（或+）：逻辑或，用于连接并列关系的检索词。
  表示查找含有检索词 A 和 B 之一，或同时包含检索词 A 和 B 的信息 ：A　OR　B
- NOT（或-）：逻辑非，排除不需要的和影响检索结果的概念。
  表示含有检索词 A 并且不含有检索词 B 的信息：A　NOT　B

运算符之间的优先级：NOT > AND > OR，如检索表达式：中国　NOT　日本　AND　歌曲　OR　小说，搜索结果为：名字包含中国但是不包含日本的歌曲或者小说。

利用小括号“()”可以设置个性化的检索方程，例如检索出不包含日本在内的有关教育或法律方面的大学：

```
(university OR college) AND (education OR Law) NOT Japan
```

对表 1-5 的文档集建立单词－文档矩阵，如果单词在文档中出现则记为 1，单词没有在文档中出现则记为 0，结果如表 1-6 所示。

表 1-6　单词-文档矩阵

	doc1	doc2	doc3	doc4
人工	1	1	1	0
智能	1	1	1	0
成为	1	0	0	0
互联网	1	0	1	0
大会	1	0	0	0
焦点	1	0	0	0
谷歌	0	1	0	1
推出	0	1	0	0
开源	0	1	0	1
系统	0	1	0	0
工具	0	1	0	1
的	0	0	1	0
未来	0	0	1	0
在	0	0	1	0
机器	0	0	0	1
学习	0	0	0	1

单词-文档矩阵从行来看，每一行是一个行向量，对应每个词项的文档向量，表示该词项在哪些文档中出现，在哪些文档中不出现；从列来看，每一列是一个列向量，对应每个文档的词项向量，表示该文档中哪些词项出现了，哪些词项没有出现。

如果想要查询包含“谷歌”“开源”但不包含“大会”的文档，构造布尔查询：

```
谷歌 AND 开源 NOT 大会
```

分别取出“谷歌”“开源”以及“大会”对应的行向量，对“大会”对应的行向量取反算：

```
谷歌：	0	1	0	1
开源：	0	1	0	1
大会：	1	0	0	0 (取反:0	1	1	1)
```

然后进行与运算：

```
0101 AND  0101 AND 0111=0101
```

结果向量中第 2 和第 4 个元素为 1，文档 2 和文档 4 是符合查询条件的结果。

布尔检索模型简单直观，有以下优点：

第一，与人们的思维习惯一致：用户可以通过布尔逻辑运算符“AND”“OR”“NOT”将用户的提问“翻译”成系统可接受的形式。

第二，布尔逻辑式表达直观清晰。

第三，方便用户进行扩检和缩检：用户可通过增加逻辑“与”进行缩小检索，增加逻辑“或”进行扩展检索。

第四，易于计算机实现：由于布尔检索是以比较方式在集合中进行检索的，返回结果只有 1 和 0，易于实现，这也是现在的各种检索系统中都提供布尔检索的重要原因。

布尔检索模型的缺点：

第一，它的检索策略只基于 0 和 1 二元判定标准。例如，一篇文档只有相关和不相关两种状态，缺乏文档分级（rank）的概念，不能进行关键词重要性排序，限制了检索功能。

第二，没有反映概念之间内在的语义联系。所有的语义关系被简单的匹配代替，常常很难将用户的信息需求转换为准确的布尔表达式。

第三，完全匹配会导致太少的结果文档被返回。没有加权的概念，容易出现漏检。

1.5　tf-idf 权重计算

tf-idf 中文称为词频-逆文档频率，用以计算词项对于一个文档集或一个语料库中的一份文件的重要程度。词项的重要性随着它在文档中出现的次数成正比增加，但同时会随着它在文档集中出现的频率成反比下降。换句话说，如果一个词项在一篇文档中出现的频率非常高，说明其重要性比较高，但是如果这个词项在文档集中的其他的文档中出现的频率也很高，那么说明这个词语有可能是比较通用比较常见的。

tf（term frequency）代表词项频率，要想计算一份文档中某个词的词频，统计该词在整篇文档中出现的次数即可。文档有长短之分，举个例子，一篇 3000 字的文章中词语“足球”出现了 3 次，我们很难断定这篇文章就是和足球相关的，但是一篇 140 字的微博中同样出现三次“足球”，基本可以断定微博内容和足球有关。为了削弱文档长度的影响，需要将词频标准化，计算方法如下：

$$\text{词频}(tf_{t,d}) = \frac{\text{单词在文档中出现的次数}}{\text{文档的总词数}}$$

另外，词频标准化的方法不止一种，Lucene 中采用了另外一种词频标准化方法：

$$\text{词频}(tf_{t,d}) = \sqrt{\text{单词在文档中出现的次数}}$$

文档频率用 df（document frequency）表示，代表文档集中包含某个词的所有文档数目。df 通常比较大，把它映射到一个较小的取值范围，用逆文档频率（inverse document frequency，缩写为 idf）来表示：

$$\text{逆文档频率}(idf_t) = \log\left(\frac{\text{文档集总的文档数}}{\text{包含某个词的文档数}+1}\right) = \log\left(\frac{N}{df_t+1}\right)$$

上式中分母越大，说明该词越常见，逆文档频率越小。分母中文档数加 1 是进行平滑处

理，防止所有文档都不包含某个词时分母为 0 的情况发生。词项的权重用 TF-IDF 来表示，计算公式如下：

$$\mathrm{tf}-\mathrm{idf}=\text{词频}(tf_{t,\ d})*\text{逆文档频率}(idf_t)$$

通过 tf-idf 可以把文档表示成 n 维的词项权重向量：

$$document\ vector=(W_1,\ W_2,\ \ldots W_n)$$

计算词项的 tf-idf 的 Java 代码如代码清单 1-1 所示。在 TfIdfCal 类中依次定义了计算词项频率 tf、文档频率 df、逆文档频率 idf、tf-idf 的方法，在 main 方法中以表 1-4 中的文档作为测试的文档集合，最后依次输出“谷歌”的词项频率、文档频率和 tf-idf 值。

代码清单 1-1

```
import java.util.Arrays;
import java.util.List;
public class TfIdfCal {
    public double tf(List<String> doc, String term) {
        double termFrequency = 0;
        for (String str : doc) {
            if (str.equalsIgnoreCase(term)) {
                termFrequency++;
            }
        }
        return termFrequency / doc.size();
    }

    public int df(List<List<String>> docs, String term) {
        int n = 0;
        if (term != null && term != "") {
            for (List<String> doc : docs) {
                for (String word : doc) {
                    if (term.equalsIgnoreCase(word)) {
                        n++;
                        break;
                    }
                }
            }
        } else {
            System.out.println("term 不能为 null 或者空串");
        }
        return n;
    }
```

```
    public double idf(List<List<String>> docs, String term) {
        return  Math.log(docs.size()/(double)df(docs,term)+1);
    }

    public double tfIdf(List<String> doc, List<List<String>>
        docs, String term){
        return tf(doc, term) * idf(docs, term);
    }

    public static void main(String[] args) {
       List<String> doc1 = Arrays.asList("人工", "智能", "成为",
                            "互联网", "大会", "焦点");
       List<String> doc2 = Arrays.asList("谷歌", "推出", "开源",
                            "人工", "智能", "系统", "工具");
       List<String> doc3 = Arrays.asList("互联网", "的", "未来",
                            "在", "人工", "智能");
       List<String> doc4 = Arrays.asList("谷歌", "开源", "机器",
                            "学习", "工具");
       List<List<String>> documents = Arrays.asList(doc1, doc2,
                            doc3,doc4);
       TfIdfCal calculator = new TfIdfCal();
       System.out.println(calculator.tf(doc2, "谷歌"));
       System.out.println(calculator.df(documents, "谷歌"));
       double tfidf = calculator.tfIdf(doc2, documents, "谷歌");
       System.out.println("TF-IDF (谷歌) = " + tfidf);
    }
}
```

1.6　向量空间模型

向量空间模型（Vector Space Model，VSM）在上世纪 70 年代由信息检索领域奠基人 Salton 教授提出，并成功地应用于著名的 SMART 文本检索系统。把对文本内容的处理简化为向量空间中的向量运算，它以空间上的相似度表达语义的相似度，直观易懂。当文档被表示为文档空间的向量，就可以通过计算向量之间的相似性来度量文档间的相似性。向量空间模型的数学理论基础是余弦相似性理论，下面我们先从数学推导上认识余弦相似性理论。

在同一个 N 维空间中存在两个非零向量 A 和 B：

向量 A 记作 A=(x_1，x_2，x_3，x_4…$\cdots x_{n-2}$，x_n)
向量 B 记作 B=(y_1，y_2，y_3，y_4…… y_{n-1}，y_n)

向量 A 和 B 的夹角为 θ，那么由夹角公式可得：

$$\cos\theta = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} x_i * y_i}{\sqrt{\sum_1^n (x_i)^2} * \sqrt{\sum_1^n (y_i)^2}}$$

夹角余弦值 cosθ 是与向量的长度无关的，仅仅与向量的指向方向相关。当 A 向量和 B 向量方向完全相同时，那么两个向量之间的夹角为 0，cosθ = 1；当 A 向量和 B 向量方向完全相反时，那么两个向量之间的夹角为 180 度，cosθ = −1；当 A 向量和 B 向量互相垂直时，即夹角为 90 度，cosθ = 0。0 度角的余弦值是 1，而其他任何角度的余弦值都不大于 1，并且其最小值是−1，从而通过两个向量之间的角度的余弦值确定两个向量是否大致指向相同的方向。用向量夹角余弦值来衡量相似度：Sim(A，B) = cosθ

余弦相似度通常用于正空间，因此给出的值为 0 到 1 之间。如图 1-3 所示，假设有两个文档 doc1 和 doc2，doc1 和 doc2 在向量空间中分别用向量 d_1 和向量 d_2 来表示。

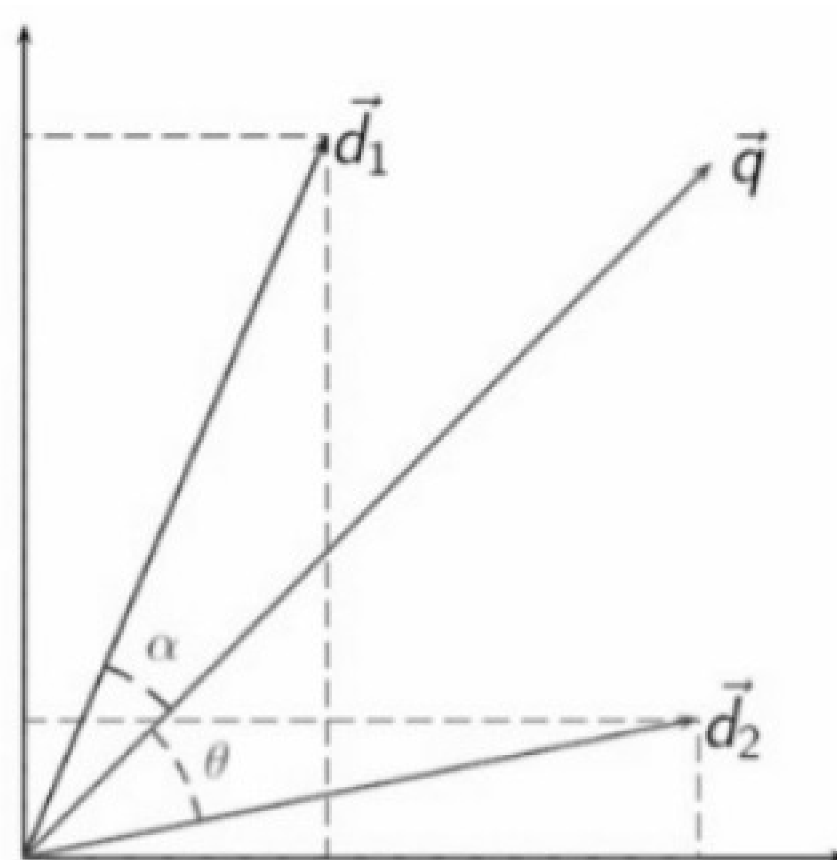


图 1-3 文档向量和查询向量

关键词查询向量为 q，d_1 和 q 的夹角为 α，d_2 和 q 的夹角为 θ。通过计算 cos(d_1，q)得出查询向量和 doc1 之间的相似性：

$$\cos\alpha = \frac{d_1 \cdot q}{|d_1|\cdot|q|}$$

计算 cos（d_2，q）得出查询向量和 doc2 之间的相似性：

$$\cos\theta = \frac{d_2 \cdot q}{|d_2|\cdot|q|}$$

比较 cosα 和 cosθ 的大小可以得出文档 1 和文档 2 哪个和查询关键词相关度更大。余弦相似性理论除了应用在信息检索模型中以外，在文本挖掘领域可用于文件比较，在数据挖掘领域中可以用来度量集群内部的凝聚力，在推荐系统中可以用来比较用户偏好的相似性。相对于标准布尔模型，向量空间模型具有如下优点：

- 基于线性代数的简单模型。
- 词组的权重不是二元的。
- 文档和查询之间的相似度取值是连续的。
- 允许根据文档间可能的相关性来进行排序。
- 允许局部匹配。

向量空间模型的 Java 实现计算见代码清单 1-2。在 Vsm 类中实现了一个静态的方法用于计算两个向量的夹角，传入参数为两个 Map，在 main 函数中给出了测试案例。

代码清单 1-2

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

public class Vsm {
    public static double calCosSim(Map<String, Double> v1,
        Map<String, Double> v2){
        double sclar = 0.0,norm1=0.0,norm2=0.0,similarity=0.0;
        Set<String> v1Keys = v1.keySet();
        Set<String> v2Keys = v2.keySet();
        Set<String> both= new HashSet<>();
        both.addAll(v1Keys);
        both.retainAll(v2Keys);
        System.out.println(both);
        for (String str1 : both) {
            sclar += v1.get(str1) * v2.get(str1);
        }
        for (String str1:v1.keySet()){
            norm1+=Math.pow(v1.get(str1),2);
        }
        for (String str2:v2.keySet()){
            norm2+=Math.pow(v2.get(str2),2);
        }
        similarity=sclar/Math.sqrt(norm1*norm2);
        System.out.println("sclar:"+sclar);
        System.out.println("norm1:"+norm1);
        System.out.println("norm2:"+norm2);
        System.out.println("similarity:"+similarity);
        return similarity;
    }
    public static void main(String[] args) {
        Map<String, Double> m1 = new HashMap<>();
        m1.put("Hello", 1.0);
        m1.put("css", 2.0);
        m1.put("Lucene", 3.0);

        Map<String, Double> m2 = new HashMap<>();
```

```
        m2.put("Hello", 1.0);
        m2.put("Word", 2.0);
        m2.put("Hadoop", 3.0);
        m2.put("java", 4.0);
        m2.put("html", 1.0);
        m2.put("css", 2.0);
        calCosSim(m1, m2);
    }
}
```

事实上，Lucene 中的评分机制更加复杂，糅合了索引期和查询期用户赋予的权重等多种因素。实际的评分公式如下：

$$\mathrm{Score}(\mathrm{q},\mathrm{d}) = \mathrm{coord}(\mathrm{q},\mathrm{d}) * \mathrm{queryNorm}(\mathrm{q}) * \sum_{t\ in\ q}(tf * idf(t)^2 * t.getBoost() * norm(t,d))$$

- coord(q,q)：评分因子，其值为出现查询词项占文档的百分比，一个文档中出现查询词项的个数越高，文档匹配程度越高。
- querynorm(q)：查询的标准因子，目的是使不同的查询之间可比较，所有的排序文档都会乘以这个因子，因此不会影响文档的排序。
- tf：文档频率。
- idf：逆文档频率。
- t.getboost()：查询时期的附加权重。
- norm(t,d)：索引时期的权重和长度因子。

整体而言，上述公式仍然是基于 tf-idf 和向量空间模型的相似度计算。但是向量空间模型仍然存在一些缺点，比如长文档的 tf 一般会越高，呈正相关性，对短文档不公平，而且查询词之间并不是完全独立的。基于此，排序模型也在不断改进和完善之中。

1.7　概率检索模型

概率检索模型从概率排序原理推导而来，是一种直接对用户需求相关性进行建模的方法，其基本思想是：给定一个查询，返回的文档能够按照查询和用户需求的相关性得分高低来排序。目前最成功的概率检索模型是 BM25（Best Match 25）模型，发展于 1970 年到 1980 年之间，目前很多商业搜索引擎使用的都是 1994 年在 BM25 的基础上进行改进的 Okapi BM25 模型。下面从概率基础推导 BM25 的评分公式。

1.7.1　贝叶斯决策理论

概率检索模型的数学基础是贝叶斯决策理论，推导 BM25 模型的评分公式先从贝叶斯公式开始。我们知道，概率是对随机事件发生的可能性的度量，取值为 0~1，比如抛一枚硬币，正面朝上的可能性和反面朝上的可能性都是 0.5。条件概率是指在某些前提条件下的概率问题，

在事件 A 发生的前提下事件 B 发生的概率记为 $P(B|A)$。联合概率是指两个事件同时发生的概率，事件 A 和事件 B 相互独立，那么 A 和 B 同时发生的概率记为 $P(AB)$，显然 $P(AB)=P(BA)$。A 和 B 同时发生的概率等于事件 A 发生的前提下事件 B 发生的概率，即：

$$P(AB) = P(B|A)P(A)$$

A 和 B 同时发生的概率也等于事件 B 发生的前提下事件 A 发生的概率，即：

$$P(AB) = P(A|B)P(B)$$

$P(AB)=P(BA)$，所以：

$$P(B|A)P(A) = P(A|B)P(B)$$

两边同时除以 $P(B)$，可得：

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

上式即为贝叶斯公式，贝叶斯决策理论在机器学习、自然语言处理等领域被广泛应用，在海量数据的文本分类问题（比如垃圾邮件、垃圾短信的甄别和过滤）上能取得非常好的效果，其核心思想是选择高概率对应的类别。

以图 1-3 为例，数据集中有实心圆和空心圆两类数据，假设数据集的统计参数已知，给出一个新的数据点 $A(x, y)$，$P_1(x, y)$表示点 A 属于实心圆的概率，$P_2(x, y)$表示点 A 属于空心圆的概率，如果 $P_1(x, y) > P_2(x, y)$，那么点 A 极有可能属于实心圆，反之属于空心圆。

概率检索模型把用户查询和要查询的文档集作为一个贝叶斯分类问题，对于任意的一个查询，文档集可以划分为与查询相关和与查询不相关两类。对于文档 D，$P(R|D)$代表文档属于相关文档集的概率，$P(NR|D)$代表文档属于不相关文档集的概率，如果 $P(R|D)>P(NR|D)$，可以认为文档 D 和用户查询相关，反之不相关，如图 1-4 所示。

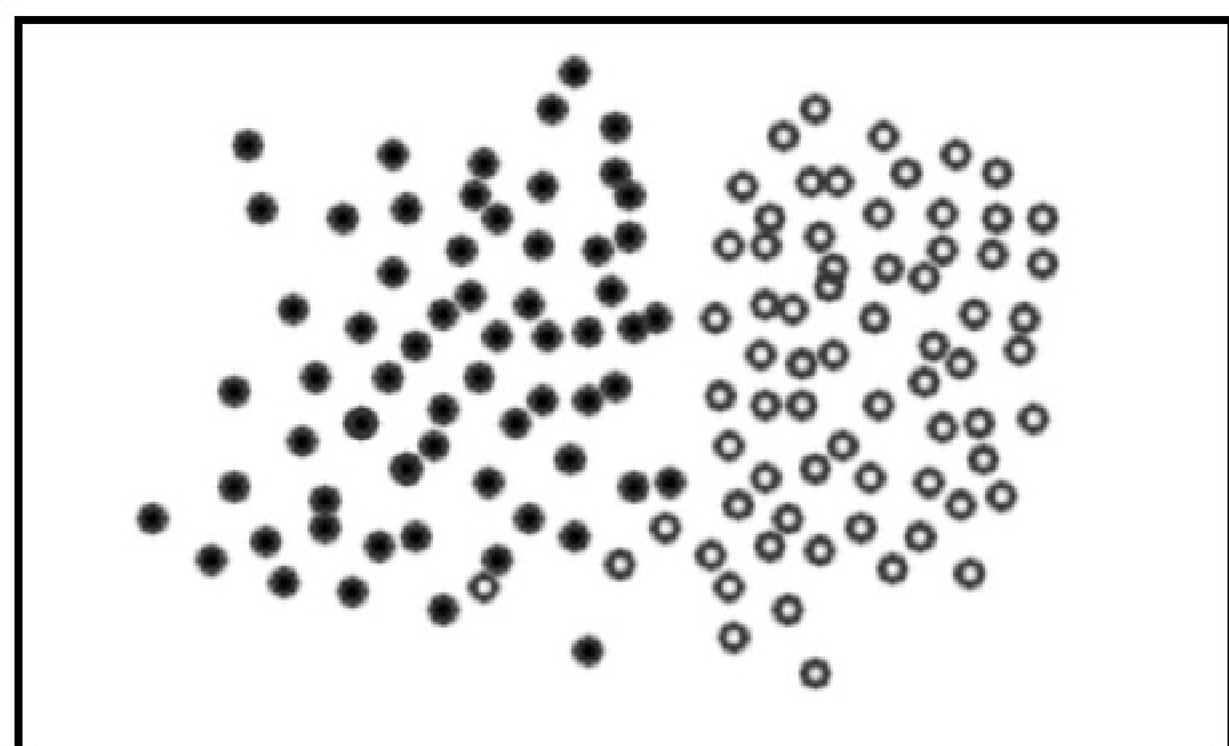


图 1-3　数据分布图

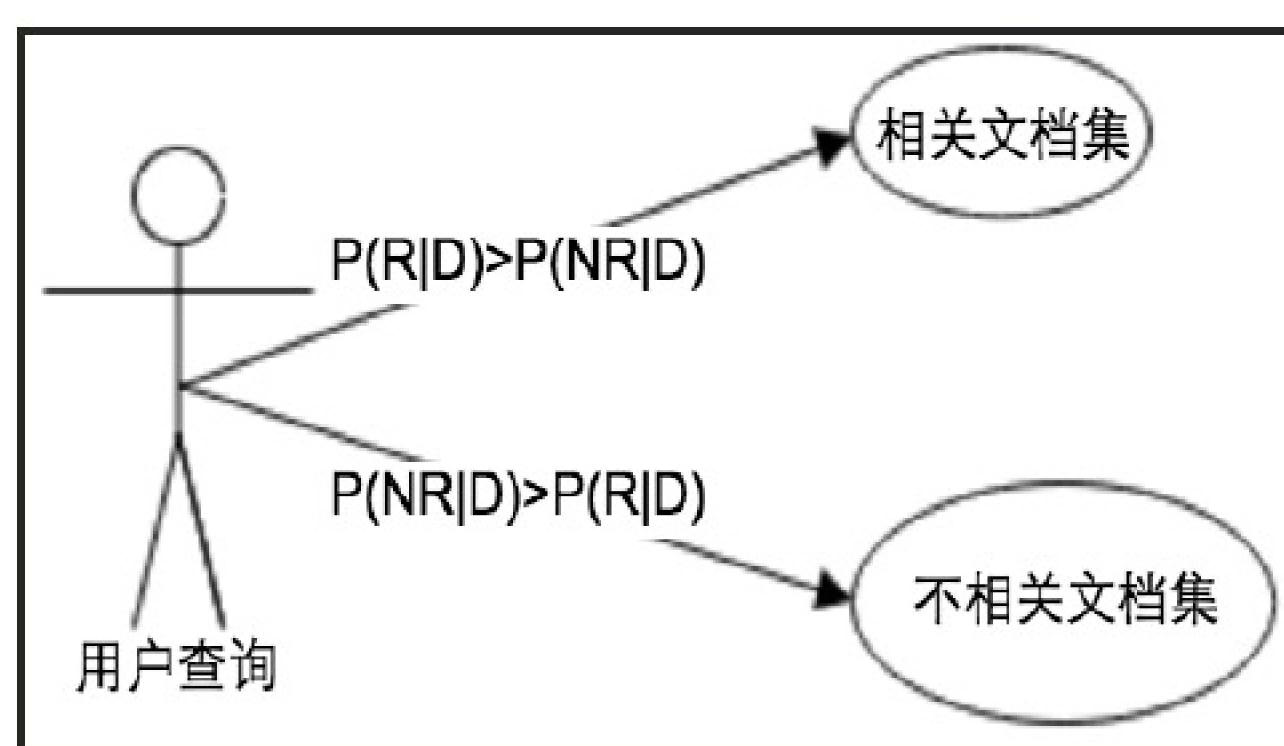


图 1-4　贝叶斯分类

问题的关键是如何比较 $P(R|D)$和 $P(NR|D)$的大小，由贝叶斯公式可得：

$$P(R|D) = \frac{P(D|R)P(R)}{P(D)}$$

$$P(NR|D) = \frac{P(D|NR)P(NR)}{P(D)}$$

比较 $P(R|D)>P(NR|D)$，等价于：

$$\frac{P(D|R)P(R)}{P(D)} > \frac{P(D|NR)P(NR)}{P(D)}$$

由于 $P(D)$是相同的，左右两边消去，等价于 $P(D|R)P(R)>P(D|NR)P(NR)$，左右两边同时再除以 $P(D|NR)P(R)$，转换为如下形式：

$$\frac{P(D|R)}{P(D|NR)} > \frac{P(NR)}{P(R)}$$

在搜索排序的时候，系统只需要将$\frac{P(D|R)}{P(D|NR)}$降序排序即可，问题进一步转换为如何计算 $P(D|R)$与 $P(D|NR)$。

1.7.2　二值独立模型

二值独立模型（Binary Independence Model，简称 BIM）也是一种概率检索模型，通过做出一些假设估算文档或者查询的相似性概率。二值独立模型中的二值是指文档和查询都表示成词项出现与否的布尔向量，词项出现记为 1，词项不出现记为 0。独立是指假设词项在文档中的出现是相互独立的，通过对词项的独立性假设可以用数学的方法描述文本，把文档频率转换为词项概率的乘积，即

$$P(D|R) = P\big((w_1, w_2, \ldots, w_n)\big|R\big) = \prod_{w\ in\ D} P(w|R)$$

假设查询中有 5 个关键词，文档 D 中只出现了第 1 个、第 3 个、第 5 个，那么通过二值假设，D 可以表示为{1,0,1,0,1}，用 P_i 表示第 i 个单词出现在相关文档集合中的概率，相关文档集合中出现文档 D 的概率可表示为：

$$P(D|R) = P_1 * (1 - P_2) * P_3 * (1 - P_4) * P_5$$

同样，对于假设 S_i 代表第 i 个单词在不相关文档集合内出现的概率，不相关文档集合中出现文档 D 的概率可表示为：

$$P(D|NR) = S_1 * (1 - S_2) * S_3 * (1 - S_4) * S_5$$

计算$\frac{P(D|R)}{P(D|NR)}$：

$$\frac{P(D|R)}{P(D|NR)} = \frac{P_1 * (1 - P_2) * P_3 * (1 - P_4) * P_5}{S_1 * (1 - S_2) * S_3 * (1 - S_4) * S_5}$$

一般情况下，$\frac{P(D|R)}{P(D|NR)}$的计算公式如下：

$$\frac{P(D|R)}{P(D|NR)} = \prod_{i:\mathrm{Di}=1} \frac{P_i}{S_i} * \prod_{i:\mathrm{Di}=0} \frac{1 - P_i}{1 - S_i}$$

其中 i:Di=1 表示单词在文档 D 中出现，i:Di=0 表示单词在文档 D 中不出现。进一步进行等价变换，其中$\left(\prod_{i:Di=1}\frac{1-P_i}{1-S_i} * \prod_{i:Di=0}\frac{1-P_i}{1-S_i}\right) = 1$：

$$\frac{P(D|R)}{P(D|NR)}=\left(\prod_{i:Di=1}\frac{P_i}{S_i}*\prod_{i:Di=1}\frac{1-S_i}{1-P_i}\right)*\left(\prod_{i:Di=1}\frac{1-P_i}{1-S_i}*\prod_{i:Di=0}\frac{1-P_i}{1-S_i}\right)$$

最终得到计算结果：

$$\frac{P(D|R)}{P(D|NR)}=\prod_{i:Di=1}\frac{P_i(1-S_i)}{S_i(1-P_i)}$$

等式两边同时取对数得到相关性计算公式：

$$\sum_{i:Di=1}\log\frac{P_i(1-S_i)}{S_i(1-P_i)}$$

相关性公式中只有 P_i 和 S_i 未知，P_i 和 S_i 分别代表第 i 个单词在相关文档集合中出现和不出现的概率，对于一个已知的查询，总的文档集中要么是和该查询相关的，要么是和该查询不相关的，根据相关与否文档集可分为两类；同时，对于一个查询，总的文档集合中的文档要么是包含查询关键词的，要么是不包含查询关键词的，根据文档的包含与否文档集也可以分为两类。综上可得表 1-7，其中总文档集合为 N，相关文档数量为 R，包含单词 i 的文档数量为 n_i，相关文档中包含单词 i 的文档为 r_i。

表1-7　文档集分类表

	相关文档集	不相关文档集	文档数量
包含单词 $i(d_i=1)$	r_i	n_i-r_i	n_i
不包含单词 $i(d_i=0)$	$R-r_i$	$(N-R)-(n_i-r_i)$	$N-n_i$
文档数量	R	$N-R$	N

如果已知 N、R、n_i、r_i，即可计算 P_i 和 S_i：

$$P_i=\frac{\text{包含单词}i\text{的相关文档数量}}{\text{总的相关文档数量}}=\frac{r_i}{R}$$

$$S_i=\frac{\text{包含单词}i\text{的不相关文档数量}}{\text{总的不相关文档数量}}=\frac{n_i-r_i}{N-R}$$

如果 r_i=0，则 P_i=0，相关性计算公式中就会出现 log(0)的情况，因此需要进行平滑处理，在分子上加上常数 0.5，分母上加上常数 1，最终：

$$P_i=\frac{(r_i+0.5)}{R+1}\quad S_i=\frac{(n_i-r_i+0.5)}{N-R+1}$$

带入相关性计算公式可得：

$$\sum_{i:q_i=d_i=1}log\frac{(r_i+0.5)/(\mathrm{R}-r_i+0.5)}{(n_i-r_i+0.5)/(N-R-n_i+r_i+0.5)}$$

上述公式即为通过二值独立模型计算用户查询和文档相关性的方法，其含义就是累加同时出现在用户查询和文档 D 中的概率，累加结果即为查询和文档的相关度。

1.7.3　Okapi BM25 模型

二值独立模型计算相关性的实际应用效果并不理想，因为二值假设只考虑了单词在文档中的出现与否，没有考虑单词的权重，BM25 模型在此基础之上进行了改进，把 idf 因子、文档长度、文档词频、查询词频等因素统统考虑进去，BM25 模型的评分公式如下：

$$\sum_{i\in Q} log\frac{(r_i+0.5)/(\mathrm{R}-r_i+0.5)}{(n_i-r_i+0.5)/(N-R-n_i+r_i+0.5)} * \frac{(k_1+1)f_i}{K+f_i} * \frac{(k_2+1)tf_{tq}}{k_2+tf_{tq}}$$

其中

$$\mathrm{K}=k_1\left[\left((1-b)+b*\frac{L_d}{L_{ave}}\right)\right]$$

对查询 Q 进行分词，依次计算每个单词在文档 D 中的分值，累加后即为查询 Q 下文档 D 的得分。上述公式分为三个部分，第一部分为二值独立模型中推导出来的相关性计算公式;第二部分是查询词在文档 D 中的权重，f_i 代表单词在文档 D 中的词频，k_1 是经验参数，K 是对文档长度的考虑，k_1 和 b 都是经验参数;公式第三部分是查询词自身的权重，tf_{tq} 是词项 t 在查询 Q 中的词频，k_2 是一个取正的调优参数，用于对查询中的词项频率进行缩放。k_1 取 0 时，公式的第二部分为 1，此时不考虑词频的因素，b 取 0 时表示忽略文档长度因素，k_2 取 0 时表示不考虑词项在查询中的权重。在没有根据开发测试集进行优化的情况下，已有的实验结果表明，参数的合理取值范围是：k_1 的取值区间为 1.2~2，b 取 0.75，k_2 取 0~1000，k_2 取值较大是因为查询一般较短，不同查询词的词频较小，较大的调节参数值可以对词频之间的差异进行放大。

1.7.4　BM25F 模型

Okapi BM25 模型提出之后被广泛应用，在计算相关性的时候只是把文档当作整体来考虑，但是并没有考虑文档不同域（也就是字段）的权重差异，结构化的数据会被切分成多个独立的域，以网页为例，网页有标题、摘要、主题词、内容等域，很显然网页标题是对一个网页内容的高度概括，标题中关键词的权重很显然要比网页内容中的关键字权重高。BM25F 在 BM25 的基础上做了一些改进，把单词在文档域中的权重得分考虑进去。BM25F 的计算公式如下：

$$\sum_{i:q_i=d_i=1} log\frac{(r_i+0.5)/(\mathrm{R}-r_i+0.5)}{(n_i-r_i+0.5)/(N-R-n_i+r_i+0.5)} * \frac{{f_i}^u}{k_1+{f_i}^u}$$

$${f_i}^u=\sum_{k=1}^{u} w_k*\frac{f_{ui}}{B_u} \qquad B_u=\left((1-b_u)+b_u*\frac{ul_u}{avgul_u}\right)$$

公式中的第一部分还是二值独立模型的评分，${f_i}^u$ 代表第 i 个单词在 u 个域中的得分之和，w_k 代表为每个域设定的权值，f_{ui} 代表第 i 个单词在第 u 个域中的词频，B_u 是第 u 个域的长度因素。在 B_u 的计算公式中，b_u 是调节因子，对于不同的域要设定不同的调节因子，ul_u 是第 u 个域的实际长度，avgul$_u$ 是文档集中这个域的平均长度。

1.8　本章小结

这一章作为本书的第一章，介绍了信息过载和信息检索的概念以及信息检索中的常用术语，之后介绍了分词的原理与分词算法，重点介绍了搜索引擎中倒排索引这种数据结构，最后介绍了检索模型中的布尔检索模型、tf-idf 词元权重计算、向量空间模型以及概率检索模型。通过本章的学习，读者应该能了解 Lucene 的数学模型，知其然也知其所以然。

第2章

Lucene 开发入门

本章学习要点：

- ✱ Lucene 简介
- ✱ Lucene 特点和架构
- ✱ Lucene 开发准备
- ✱ Lucene 分词详解
- ✱ Lucene 索引详解
- ✱ Lucene 查询详解
- ✱ Lucene 搜索高亮
- ✱ Lucene 新闻高频词提取案例

2.1 Lucene 概述

2.1.1 Lucene 简介

Lucene 是一个开源的全文检索引擎工具包，最初由 Doug Cutting 开发。早在 1997 年，资深全文检索专家 Doug Cutting 用一个周末的时间，使用 Java 语言创作了一个文本搜索的开源函数库，目的是为各种中小型应用软件加入全文检索功能。不久之后，Lucene 诞生了，2000 年 Lucene 成为 Apache 开源社区的一个子项目。随着 Lucene 被人们熟知，越来越多的用户和研发人员加入其中，完善并壮大项目的发展，Lucene 已成为最受欢迎的具有完整的查询引擎和索引引擎的全文检索库。

2.1.2 Lucene 特点

Lucene 从问世之后，引发了开源社区的巨大反响，程序员们不仅使用它构建全文检索应用，而且将之集成到各种系统软件中去，除此之外还用来构建 Web 应用。维基百科用 Lucene 建立了一个站内的强大搜索功能，用以检索站内数以千万的词条。IBM 的商业软件 Web Sphere 也采用了 Lucene 作为全文索引引擎。Lucene 以其开放源代码的特性、优异的索引结构、良好的系统架构获得了越来越多的应用。Lucene 的优点主要有以下 3 点：

1. 稳定，索引性能高

- 现代硬盘上每小时能够索引 150GB 以上的数据。
- 对内存的要求小——只需要 1MB 的堆内存。
- 增量索引和批量索引一样快。
- 索引的大小约为索引文本大小的 20%～30%。

2. 高效、准确、高性能的搜索算法

- 搜索排名——最好的结果显示在最前面。
- 许多强大的查询类型：短语查询、通配符查询、近似查询、范围查询等。
- 对字段级别搜索（如标题，作者，内容）。
- 可以对任意字段排序。
- 支持搜索多个索引并合并搜索结果。
- 支持更新操作和查询操作同时进行。
- 灵活的切面、高亮、join 和 group by 功能。
- 速度快，内存效率高，容错性好。
- 可选排序模型，包括向量空间模型和 BM25 模型。
- 可配置存储引擎。

3. 跨平台解决方案

- 作为 Apache 开源许可，在商业软件和开放程序中都可以使用 Lucene。
- 100%纯 Java 编写。
- 对多种语言提供接口。

2.1.3　Lucene 架构

先从整体上看一下 Lucene 的架构设计。图 2-1 是 Lucene 的整体架构图，是对 Lucene 精髓的概括，理解了这张图就从整体上把握住了 Lucene。

先看上层应用，首先是信息采集的过程，文件系统、数据库、万维网以及手工输入的文件都可以作为信息采集的对象，也是要搜索的文档的来源，采集万维网上的信息一般使用网络爬虫。完成信息采集之后到 Lucene 层面主要有两大任务：索引文档和搜索文档，索引文档的过程完成由原始文档到倒排索引的构建过程，搜索文档用以处理用户查询。应用层的第三部分就是用户接口，用户输入查询关键词，Lucene 完成文档搜索任务，经过分词、匹配、评分、排序等一系列过程之后返回用户想要的文档。

一次完整的搜索从用户输入要查询的关键词开始，比如想查找 Lucene 的相关学习资料，我们都会在 Google 或百度等搜索引擎中输入关键词，比如输入“Lucene，全文检索框架”，之后系统根据用户输入的关键词返回相关信息。一次检索大致可分为 4 步：

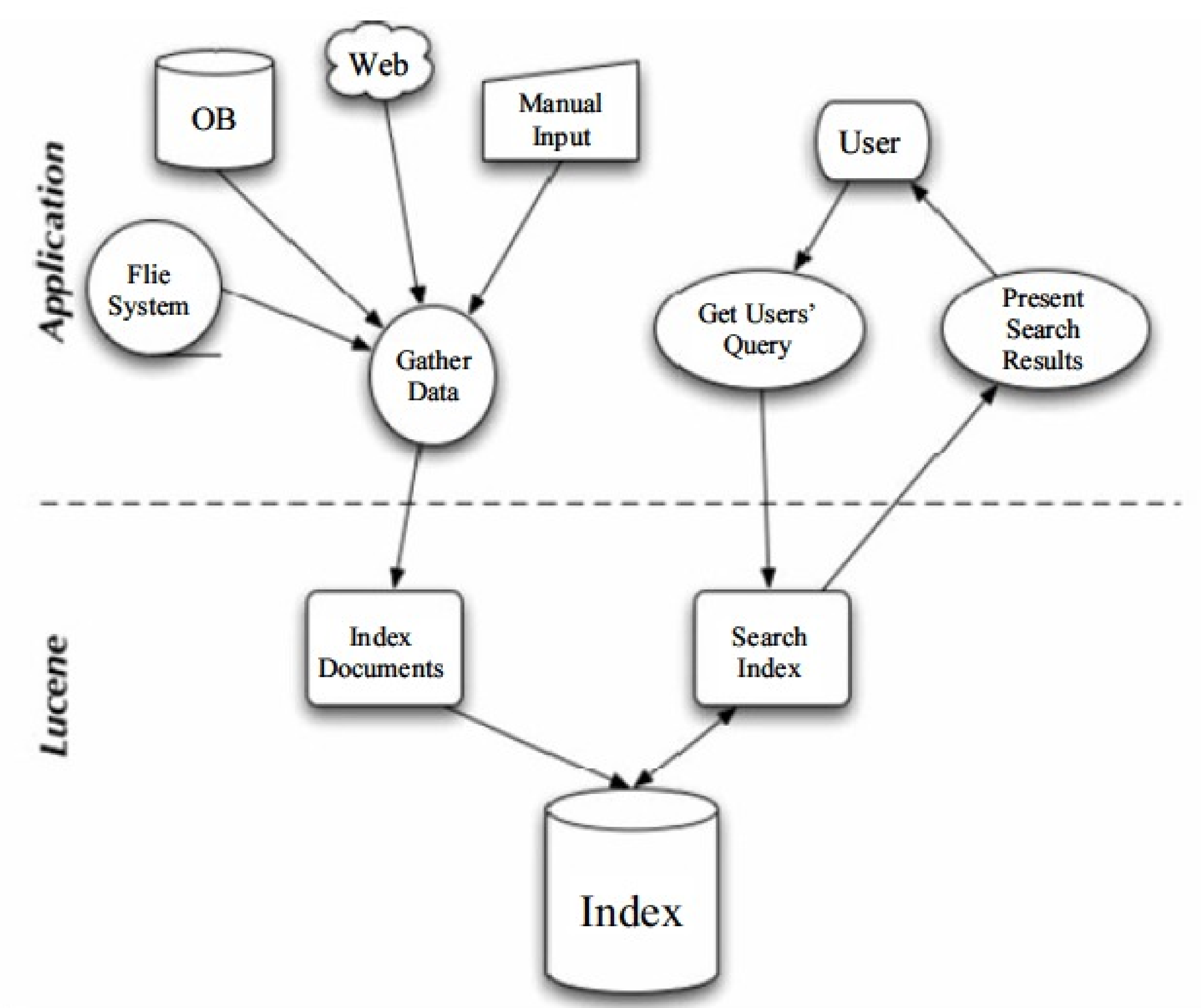


图 2-1　Lucene 架构图

第一步：查询分析

正常情况下用户输入正确的查询，比如搜索“里约奥运会”这个关键词，用户输入正确完成一次搜索，但是搜索需求通常都是全开放的，任何的用户需求都是有可能的，很大一部分还是非常口语化和个性化的，有时候还会存在拼写错误，如图 2-2 所示，用户不小心把“淘宝”打成“涛宝”，这时候需要用自然语言处理技术来做拼写纠错等处理，以正确理解用户需求。



图 2-2 搜索引擎拼写纠正

第二步：分词技术

这一步利用自然语言处理技术将用户输入的查询语句进行分词，如标准分词会把“lucene，全文检索框架”分成：lucene | 全 | 文 | 检 | 索 | 框 | 架 | ，空格分词会分成：lucene， | 全文检索框架 | ，二分法会分成：lucene | 全文 | 文检 | 检索 | 索框 | 框架 |，还有简单分词等多种分词方法。

第三步：关键词检索

提交关键词后在倒排索引库中进行匹配，倒排索引就是关键词和文档之间的对应关系，就像给文档贴上标签。比如在文档集中含有 lucene 关键词的有文档 1、文档 6、文档 9，含有

全文检索的有文档 1、文档 6，那么做与运算，同时含有 lucene 和全文检索的文档就是 1 和 6，在实际的搜索中会有更复杂的文档匹配模型。

第四步：搜索排序

对多个相关文档进行相关度计算、排序，返回给用户检索结果。

2.2　Lucene 开发准备

2.2.1　下载 Lucene 文件库

首先访问 Lucene 的官方网站（https://lucene.apache.org/），下载 Lucene 文件库，其网站页面如图 2-3 所示。

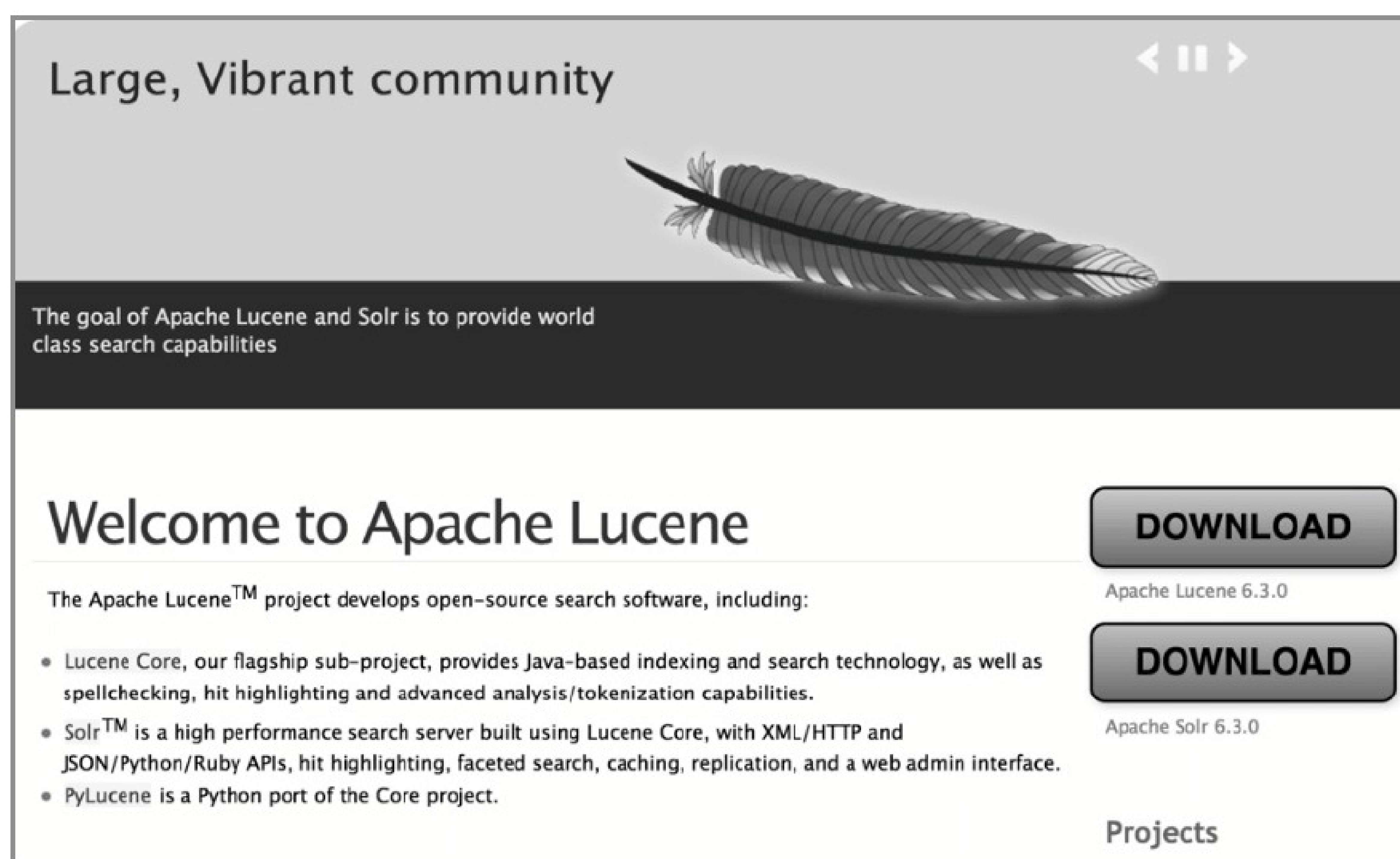


图 2-3　Lucene 下载页面

在首页即可看到一绿一红两个下载按钮，单击绿色 DOWNLOAD 按钮会跳转到 Lucene 下载页，单击红色 DOWNLOAD 会跳转到 Solr 下载页。

这里有必要对 Lucene 和 Solr 进行说明，Lucene 是一个做全文检索的库，开发者可以拿来根据实际业务需求进行使用，而 Solr 是一个基于 Lucene 的全文搜索服务器。Solr 是在 Lucene 的基础上进行扩展，并且提供了更加丰富的查询语言，可扩展性和可配置性比 Lucene 更高。除此之外 Solr 还提供了一个完善的管理界面，是一个产品级的全文搜索引擎。

官网首页提供了最新版本的下载链接，访问 http://archive.apache.org/dist/lucene/java/，可以下载 Lucene 所有的发行版本，其页面如图 2-4 所示。

本书基于 Lucene 6.0.0 版本进行讲解，下拉找到 6.0.0 文件夹，打开后界面如图 2-5 所示。

下载列表中的 lucene-6.0.0.zip（或者 lucene-6.0.0.tgz），下载完成以后不需要任何安装，解压缩即可。如果想阅读和学习 Lucene 源码，可以下载 src 版本，即 lucene-6.0.0-src.tgz。

Index of /dist/lucene/java

Name	Last modified	Size	Description
Parent Directory		-	
2.9.4/	2013-01-16 12:49	-	
3.0.3/	2011-03-30 13:18	-	
3.1.0/	2011-03-30 13:25	-	
3.2.0/	2011-06-03 14:00	-	
3.3.0/	2011-06-30 05:19	-	
3.4.0/	2011-09-13 15:41	-	
3.5.0/	2011-11-25 23:23	-	
3.6.0/	2012-04-11 20:03	-	
3.6.1/	2012-07-21 17:39	-	
3.6.2/	2013-01-16 12:44	-	
4.0.0-ALPHA/	2012-06-29 01:56	-	
4.0.0-BETA/	2012-08-10 11:26	-	
4.0.0/	2013-01-16 12:33	-	
4.1.0/	2013-01-21 20:36	-	
4.10.0/	2014-09-02 22:00	-	

图 2-4　Lucene 下载列表

Index of /dist/lucene/java/6.0.0

Name	Last modified	Size	Description
Parent Directory		-	
changes/	2016-04-07 15:06	-	
KEYS	2016-04-01 20:42	160K	
lucene-6.0.0-src.tgz	2016-04-01 20:42	28M	
lucene-6.0.0-src.tgz.asc	2016-04-01 20:42	819	
lucene-6.0.0-src.tgz.md5	2016-04-01 20:42	55	
lucene-6.0.0-src.tgz.sha1	2016-04-01 20:42	63	
lucene-6.0.0.tgz	2016-04-01 20:42	63M	
lucene-6.0.0.tgz.asc	2016-04-01 20:42	819	
lucene-6.0.0.tgz.md5	2016-04-01 20:42	51	
lucene-6.0.0.tgz.sha1	2016-04-01 20:42	59	
lucene-6.0.0.zip	2016-04-01 20:42	74M	
lucene-6.0.0.zip.asc	2016-04-01 20:42	819	
lucene-6.0.0.zip.md5	2016-04-01 20:42	51	
lucene-6.0.0.zip.sha1	2016-04-01 20:42	59	

图 2-5　Lucene 6.0 下载页面

2.2.2　工程中引入 Lucene

在工程中引入 Lucene 有两种方式，第一种是传统的手工导入 jar 包的方法，第二种是使用 maven 管理。

- 手工导入 jar 包

  以添加 Lucene 核心模块为例，找到 lucene-6.0.0/core/lucene-core-6.0.0.jar，添加到工程中即可。
- 添加 maven 依赖

  到 maven 仓库（http://mvnrepository.com/）中搜索关键词 Lucene，如图 2-6 所示。获取 Lucene 模块的 maven 坐标，添加到 maven 工程的 pom.xml 文件中即可。

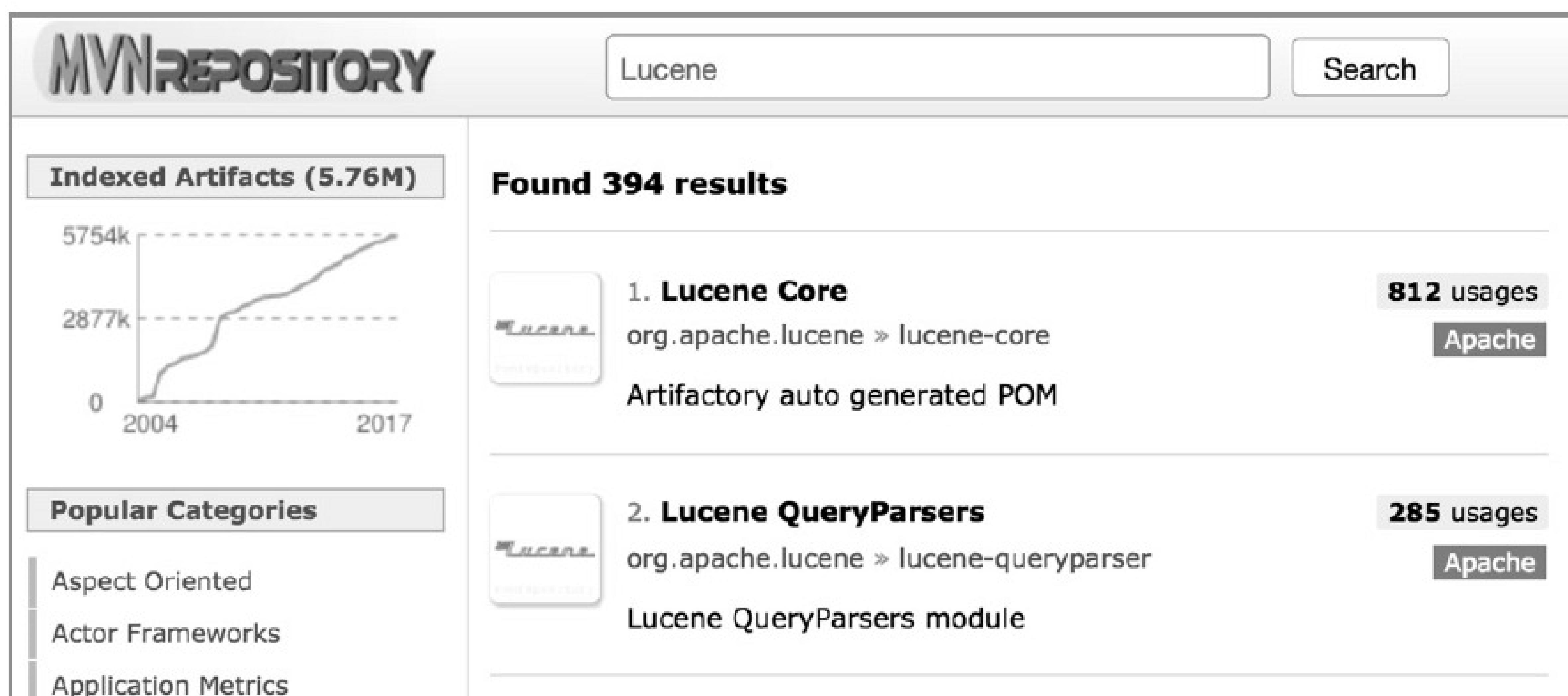


图 2-6　maven 仓库中下载 Lucene

例如，Lucene 核心模块的坐标如下：

```
<dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-core</artifactId>
    <version>6.0.0</version>
</dependency>
```

2.2.3　下载 Luke

相信大家非常熟悉关系型数据库，我们把一条条数据存入数据库中，打开数据库管理系统就能看到一条条的数据，非常直观。在全文检索系统中，数据都会被处理成为索引这种数据结构。可以不可以像数据库一样查看存放在索引中的数据呢？当然可以，Luke 就是用来查看 Lucene、Solr、Elasticsearch 索引的 GUI 工具，方便开发和诊断。Luke 的主要功能如下：

- 查看文档和分析字段内容。
- 搜索索引。
- 执行索引维护。
- 从 HDFS 读取索引。
- 将全部或部分索引转换为 XML 格式导出。
- 测试自定义的 Lucene 分词器。

需要说明的是，Luke 的版本要和 Lucene 的版本一致，比如，使用的是 6.0 版本的 Lucene 库来创建索引，那么也要使用 6.0 版本的 Luke 来查看索引。

Luke 是开源工具，代码托管在 GitHub 之上，项目地址为 https://github.com/DmitryKey/luke/releases。在浏览器中打开项目链接，找到 Luke 6.0.0，下载 luke-6.0.0-luke-release.zip 并解压缩。在 luke-6.0.0-luke-release 目录下可以看到 3 个文件：luke.bat、luke.sh 和 target。如果你使用的是 Windows 操作系统，那么可以直接单击 luke.bat 来启动 Luke；如果你使用的是 Linux 操作系统，那么可以打开终端，切换到 Luke 根目录然后运行.luke.sh 命令，之后即可成功启动 Luke。如果你看到如图 2-7 所示的界面，说明 Luke 启动成功。

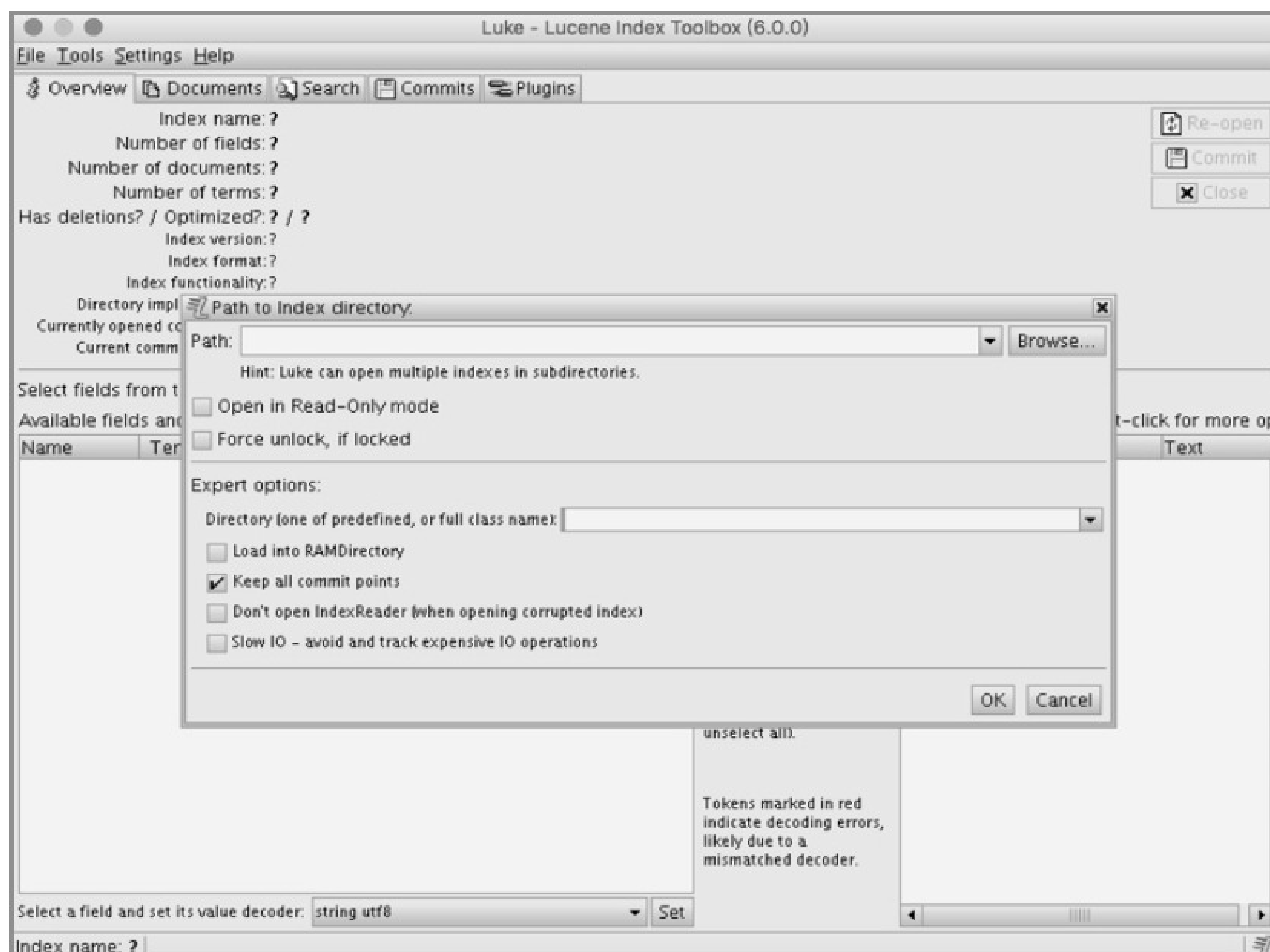


图 2-7　Luke 启动界面

2.2.4　下载 IK 分词工具

IK Analyzer（分词器）是一个开源的、基于 Java 语言开发的轻量级中文分词工具包。从 2006 年 12 月推出 1.0 版开始，IK Analyzer 已经推出了 4 个大版本。最初它是以开源项目 Luence 为应用主体的，结合词典分词和文法分析算法的中文分词组件。从 3.0 版本开始，IK Analyzer 发展为面向 Java 的公用分词组件，独立于 Lucene 项目，同时提供了对 Lucene 的默认优化实现。在 2012 版本中，IK Analyzer 实现了简单的分词歧义排除算法，标志着 IK 分词器从单纯的词典分词向模拟语义分词衍化。IK Analyzer 2012 有以下特性：

（1）采用了特有的“正向迭代最细粒度切分算法”，支持细粒度和智能分词两种切分模式。

（2）在系统环境：Core2 i7 3.4G 双核，4G 内存，Window 7 64 位，Sun JDK 1.6_29 64 位普通 PC 环境测试，IK 2012 具有 160 万字/秒（3000KB/S）的高速处理能力。

（3）2012 版本的智能分词模式支持简单的分词排歧义处理和数量词合并输出。

（4）采用了多子处理器分析模式，支持英文字母、数字、中文词汇等分词处理，兼容韩文、日文字符。

（5）优化的词典存储，更小的内存占用。支持用户词典扩展定义。特别的，在 2012 版本，词典支持中文、英文、数字混合词语。

以下是 IK Analyzer 支持细粒度切分和智能切分两种切分方式的演示样例。

文本 1：中华人民共和国国歌

智能分词结果：中华人民共和国|国歌|

最细粒度分词结果：中华人民共和国|中华人民|中华|华人|人民共和国|人民|共和国|共和|国|国歌|

文本 2：王老师说的确实很有道理

智能分词结果：王老师|说的|确实|很有|道理|

最细粒度分词结果：王老师|老师|师说|说的|的确|的|确实|很有|有道|有|道理|

IK Analyzer 下载地址为：https://code.google.com/archive/p/ik-analyzer/downloads，打开链接后，下载最近更新的 IK Analyer 2012 upgrade 6 源码包，即 IK Analyzer 2012_u6_source.rar，解压之后的安装包主要包含下列文件：

- IKAnalyzer 中文分词器使用手册
- IKAnalyzer2012.jar（主 jar 包）
- IKAnalyzer.cfg.xml（分词器扩展配置文件）
- stopword.dic（停止词典）
- LICENSE.TXT，NOTICE.TXT（Apache 版权申明）
- DOC 文件夹（API 说明文档）

IK Analyzer 的安装部署十分简单，主要有以下两步：

步骤 01 把 IKAnalyzer2012.jar 部署于项目的 lib 目录中。

步骤 02 把 IKAnalyzer.cfg.xml 与 stopword.dic 文件放置在 class 根目录（对于 Web 项目，通常是 WEB-INF/classes 目录，同 hibernate、log4j 等配置文件相同）下即可。

开发之前下载好 Lucene 6.0.0、Luke 6.0.0 和 IK Analyzer 2012，准备工作就完成了。

2.2.5　工程搭建

本章关于 Lucene 的案例代码都放在一个 Java 工程中，工程目录结构如图 2-8 所示。

indexdir 文件夹用于存放 Lucene 索引，lib 文件夹用于存放 jar 包，src 目录下有 3 个包用来存放 Java 类。tup.lucene.analyzer 目录下的 IkVSSmartcn.java 用于对比 IK 分词器和 Lucene 自带的中文智能分词器，StdAnalyzer.java 用于测试标准分词器，VariousAnalyzers.java 用于测试多种分词器；tup.lucene.ik 目录下的 IKTokenizer6x.java 和 IKAnalyzer6x.java 用于配置 IK 分词器；tup.lucene.index 目录下的 CreateIndex.java 用来创建索引；tup.lucene.queries 目录下的 QueryIndex.java 用于查询索引。

读者可以按照图 2-8 所示的目录结构新建空白 Java Project 并构建好工程目录，Java 类可以随着后面的学习逐步添加。需要添加 Lucene 的 jar 包如表 2-1 所示，在我们准备好的 Lucene 安装包和 IK Analyzer 安装包中按照表中的路径找到并拷贝到 lib 文件夹下即可。

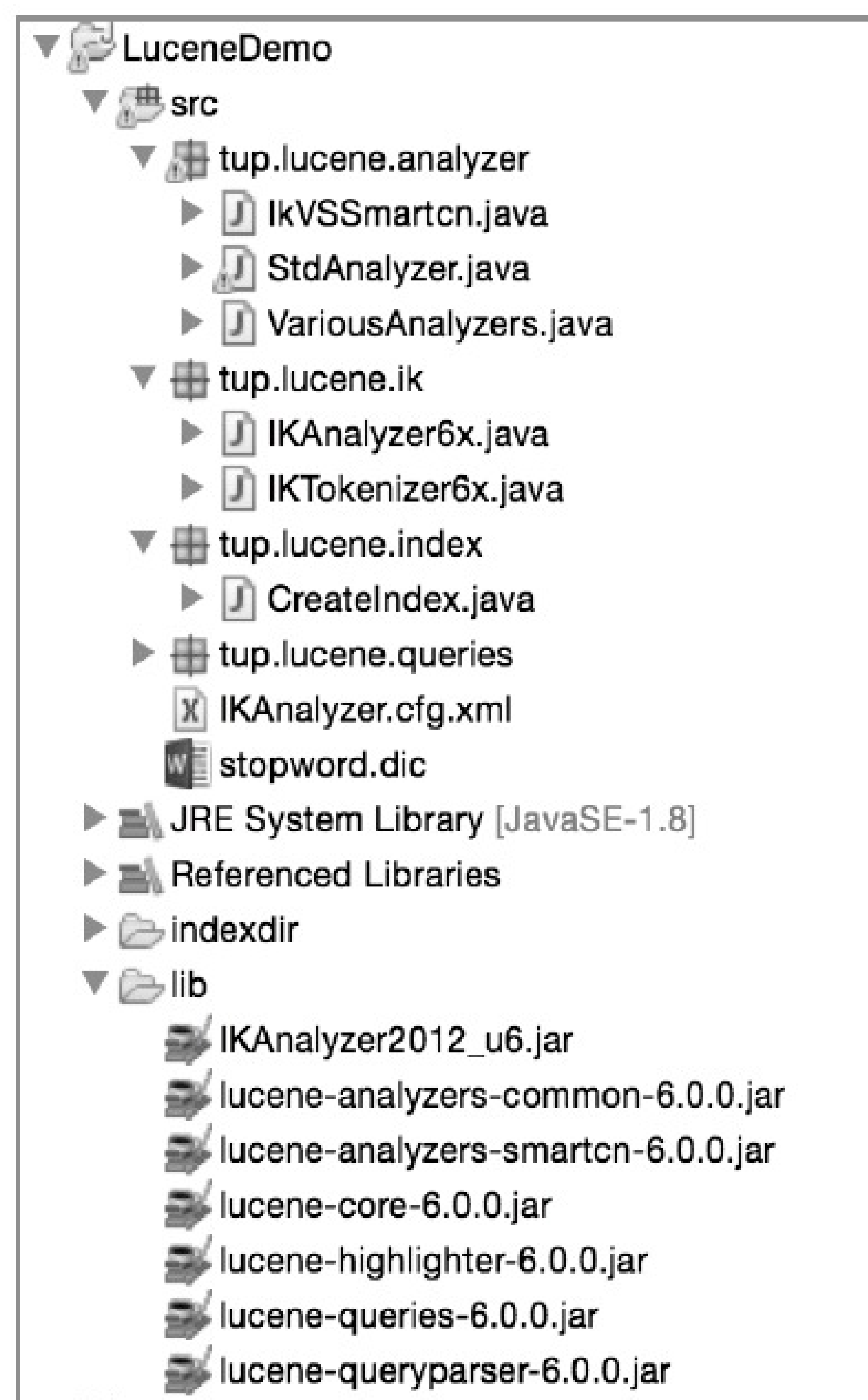


图 2-8　LuceneDemo 工程目录

表 2-1　jar 包位置对照表

jar 包名	位　置
lucene-core-6.0.0.jar	lucene-6.0.0/core/
lucene-analyzers-common-6.0.0.jar	lucene-6.0.0/analysis/common/
lucene-analyzers-smartcn-6.0.0.jar	lucene-6.0.0/analysis/smartcn/
lucene-highlighter-6.0.0.jar	lucene-6.0.0/highlighter/
lucene-queries-6.0.0.jar	lucene-6.0.0/queries/
lucene-queryparser-6.0.0.jar	lucene-6.0.0/queryparser/
IKAnalyzer2012_u6.jar	IKAnalyzer2012_u6/
lucene-memory-6.0.0.jar	lucene-6.0.0/memory

2.3　Lucene 分词详解

2.3.1　Lucene 分词系统

在第 1 章中已经提到，索引和查询都是以词项为基本单位，词项是词条化的结果。在 Lucene 中，分词主要依靠 Analyzer 类解析实现。Analyzer 类是一个抽象类（public abstract class org.apache.lucene.analysis.Analyzer），切分词的具体规则是由子类实现的，所以对于不同的语言规则，要有不同的分词器。

Analyzer 内部主要通过 TokenStream 类实现。Tonkenizer 类和 TokenFilter 类是 TokenStream 的两个子类。Tokenizer 处理单个字符组成的字符流，读取 Reader 对象中的数据，处理后转换成词汇单元。TokenFilter 完成文本过滤器的功能，但在使用过程中必须注意不同过滤器的使用顺序。

在创建索引的时候需要用到分词器，在进行索引查询的时候也会用到分词器，并且这两个地方要使用同一个分词器，否则可能会搜索不出来结果。Lucene 提供了多种分词方法，简介如下：

- StopAnalyzer（停用词分词器）
  StopAnalyzer 能过滤词汇中的特定字符串和词汇，并且完成大写转小写的功能。
- StandardAnalyzer（标准分词器）
  StandardAnalyzer 根据空格和符号来完成分词，还可以完成数字、字母、E-mail 地址、IP 地址以及中文字符的分析处理，还可以支持过滤词表，用来代替 StopAnalyzer 能够实现的过滤功能。
- WhitespaceAnalyzer（空格分词）
  WhitespaceAnalyzer 使用空格作为间隔符的词汇分割分词器。处理词汇单元的时候，以空格字符作为分割符号。分词器不做词汇过滤，也不进行小写字符转换。实际中可以用来支持特定环境下的西文符号的处理。由于不完成单词过滤和小写字符转换功能，也不需要过滤词库支持。词汇分割策略上简单使用非英文字符作为分割符，不需要分词词库支持。

- SimpleAnalyzer（简单分词）
  SimpleAnalyzer 具备基本西文字符词汇分析的分词器，处理词汇单元时，以非字母字符作为分割符号。分词器不能做词汇的过滤，只进行词汇的分析和分割。输出的词汇单元完成小写字符转换，去掉标点符号等分割符。在全文检索系统开发中，通常用来支持西文符号的处理，不支持中文。由于不完成单词过滤功能，所以不需要过滤词库支持。词汇分割策略上简单使用非英文字符作为分割符，不需要分词词库的支持。
- CJKAnalyzer（二分法分词）
  内部调用 CJKTokenizer 分词器，对中文进行分词，同时使用 StopFilter 过滤器完成过滤功能，可以实现中文的多元切分和停用词过滤。
- KeywordAnalyzer（关键词分词）
  把整个输入作为一个单独词汇单元，方便特殊类型的文本进行索引和检索。针对邮政编码、地址等文本信息使用关键词分词器进行索引项建立非常方便。

2.3.2　分词器测试

Lucene 标准分词器会把句子分成一个一个单个的单词，标准分词的代码见代码清单 2-1。

代码清单 2-1　Lucene 标准分词

```
import java.io.IOException;
import java.io.StringReader;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.analysis.tokenattributes.CharTermAttribute;
public class StdAnalyzer {
    private static String strCh = "中华人民共和国简称中国，是一个有 13 亿人口
       的国家";
    private static String strEn = "Dogs can not achieve a place,
     eyes can reach; ";
    public static void main(String[] args) throws IOException {
        System.out.println("StandardAnalyzer 对中文分词:");
        stdAnalyzer(strCh);
        System.out.println("StandardAnalyzer 对英文分词:");
        stdAnalyzer(strEn);
    }
    public static void stdAnalyzer(String str) throws IOException{
        Analyzer analyzer = null;
        analyzer = new StandardAnalyzer();
        StringReader reader = new StringReader(str);
        TokenStream toStream = analyzer.tokenStream(str, reader);
        toStream.reset();
        CharTermAttribute teAttribute =
```

```
                    toStream.getAttribute(CharTermAttribute.class);
        System.out.println("分词结果：");
        while (toStream.incrementToken()) {
            System.out.print(teAttribute.toString() + "|");
        }
        System.out.println("\n");
        analyzer.close();
    }
}
```

运行结果：

```
StandardAnalyzer对中文分词：
分词结果：
中|华|人|民|共|和|国|简|称|中|国|是|一|个|有|13|亿|人|口|的|国|家|
StandardAnalyzer对英文分词：
分词结果：
dogs|can|achieve|place|eyes|can|reach|
```

下面重构以上代码，测试多种分词器的分词效果，见代码清单 2-2。

代码清单 2-2　Lucene 多种分词器示例

```
import java.io.IOException;
import java.io.StringReader;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.cjk.CJKAnalyzer;
import org.apache.lucene.analysis.cn.smart.SmartChineseAnalyzer;
import org.apache.lucene.analysis.core.KeywordAnalyzer;
import org.apache.lucene.analysis.core.SimpleAnalyzer;
import org.apache.lucene.analysis.core.StopAnalyzer;
import org.apache.lucene.analysis.core.WhitespaceAnalyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.analysis.tokenattributes.CharTermAttribute;
public class VariousAnalyzers {
    private static String str = "中华人民共和国简称中国，是一个有13亿人口的国家";
    public static void main(String[] args) throws IOException {
        Analyzer analyzer = null;
        analyzer = new StandardAnalyzer(); // 标准分词
        System.out.println("标准分词:" + analyzer.getClass());
        printAnalyzer(analyzer);
        analyzer = new WhitespaceAnalyzer();  // 空格分词
```

```
        System.out.println("空格分词:" + analyzer.getClass());
        printAnalyzer(analyzer);
        analyzer = new SimpleAnalyzer();  // 简单分词
        System.out.println("简单分词:" + analyzer.getClass());
        printAnalyzer(analyzer);
        analyzer = new CJKAnalyzer();  // 二分法分词
        System.out.println("二分法分词:" + analyzer.getClass());
        printAnalyzer(analyzer);
        analyzer = new KeywordAnalyzer();  // 关键字分词
        System.out.println("关键字分词:" + analyzer.getClass());
        printAnalyzer(analyzer);
        analyzer = new StopAnalyzer();  // 停用词分词
        System.out.println("停用词分词:" + analyzer.getClass());
        printAnalyzer(analyzer);
        analyzer = new SmartChineseAnalyzer();  // 中文智能分词
        System.out.println("中文智能分词:" + analyzer.getClass());
        printAnalyzer(analyzer);
    }
    public static void printAnalyzer(Analyzer analyzer)
    throws IOException {
        StringReader reader = new StringReader(str);
        TokenStream toStream = analyzer.tokenStream(str,  reader);
        toStream.reset(); // 清空流
        CharTermAttribute teAttribute = toStream.getAttribute
       (CharTermAttribute.class);
        while  (toStream.incrementToken()) {
            System.out.print(teAttribute.toString() + "|");
        }
        System.out.println("\n");
        analyzer.close();
    }
}
```

运行结果：

标准分词：

class org.apache.lucene.analysis.standard.StandardAnalyzer
中|华|人|民|共|和|国|简|称|中|国|是|一|个|有|13|亿|人|口|的|国|家|

空格分词：

class org.apache.lucene.analysis.core.WhitespaceAnalyzer
中华人民共和国简称中国，|是一个有 13 亿人口的国家|

简单分词：

```
class org.apache.lucene.analysis.core.SimpleAnalyzer
中华人民共和国简称中国|是一个有|亿人口的国家|
```

二分法分词：

```
class org.apache.lucene.analysis.cjk.CJKAnalyzer
中华|华人|人民|民共|共和|和国|国简|简称|称中|中国|是一|一个|个有|13|亿人|人口|
口的|的国|国家|
```

关键字分词：

```
class org.apache.lucene.analysis.core.KeywordAnalyzer
中华人民共和国简称中国，是一个有 13 亿人口的国家|
```

停用词分词：

```
class org.apache.lucene.analysis.core.StopAnalyzer
中华人民共和国简称中国|是一个有|亿人口的国家|
```

中文智能分词：

```
class org.apache.lucene.analysis.cn.smart.SmartChineseAnalyzer
中华人民共和国|简称|中国|是|一个|有|13|亿|人口|的|国家|
```

2.3.3　IK 分词器配置

Lucene 6.0 使用 IK 分词器需要修改 IKAnalyzer 和 IKTokenizer。在包 tup.lucene.ik 下新建一个 IKTokenizer6x 类和一个 IKAnalyzer6x 类。IKTokenizer6x 类的代码见代码清单 2-3，IKAnalyzer6x 类的代码见代码清单 2-4。

代码清单 2-3　IKTokenizer6x.java

```
import java.io.IOException;
import org.apache.lucene.analysis.Tokenizer;
import org.apache.lucene.analysis.tokenattributes
      .CharTermAttribute;
import org.apache.lucene.analysis.tokenattributes.OffsetAttribute;
import org.apache.lucene.analysis.tokenattributes.TypeAttribute;
import org.wltea.analyzer.core.IKSegmenter;
import org.wltea.analyzer.core.Lexeme;
public class IKTokenizer6x extends Tokenizer {
    // IK 分词器实现
    private IKSegmenter _IKImplement;
    // 词元文本属性
    private final CharTermAttribute termAtt;
    // 词元位移属性
```

```
    private final OffsetAttribute offsetAtt;
    // 词元分类属性
    //（该属性分类参考 org.wltea.analyzer.core.Lexeme 中的分类常量）
    private final TypeAttribute typeAtt;
    // 记录最后一个词元的结束位置
    private int endPosition;
    // Lucene 6.x Tokenizer 适配器类构造函数；实现最新的 Tokenizer 接口
    public IKTokenizer6x(boolean useSmart) {
        super();
        offsetAtt = addAttribute(OffsetAttribute.class);
        termAtt = addAttribute(CharTermAttribute.class);
        typeAtt = addAttribute(TypeAttribute.class);
        _IKImplement = new IKSegmenter(input, useSmart);
    }
    @Override
    public boolean incrementToken() throws IOException {
        clearAttributes();     // 清除所有的词元属性
        Lexeme nextLexeme = _IKImplement.next();
        if (nextLexeme != null) {
          // 将 Lexeme 转成 Attributes
         termAtt.append(nextLexeme.getLexemeText());    // 设置词元文本
         termAtt.setLength(nextLexeme.getLength());     // 设置词元长度
         offsetAtt.setOffset(nextLexeme.getBeginPosition(),
                    nextLexeme.getEndPosition());       // 设置词元位移
        //记录分词的最后位置
        endPosition = nextLexeme.getEndPosition();
        typeAtt.setType(nextLexeme.getLexemeText());    // 记录词元分类
        return true;        // 返回 true 告知还有下个词元
      }
        return false;       // 返回 false 告知词元输出完毕
    }
    @Override
    public void reset() throws IOException {
        super.reset();
        _IKImplement.reset(input);
    }
    @Override
    public final void end() {
        int finalOffset = correctOffset(this.endPosition);
        offsetAtt.setOffset(finalOffset, finalOffset);
    }
}
```

代码清单 2-4　IKAnalyzer6x.java

```
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.Tokenizer;
public class IKAnalyzer6x extends Analyzer {
    private boolean useSmart;
    public boolean useSmart() {
        return useSmart;
    }
    public void setUseSmart(boolean useSmart) {
        this.useSmart = useSmart;
    }
    public IKAnalyzer6x() {
        this(false);  // IK 分词器 Lucene Analyzer 接口实现类;
                      // 默认细粒度切分算法
    }
    // IK 分词器 Lucene Analyzer 接口实现类；当为 true 时，分词器进行智能切分
    public IKAnalyzer6x(boolean useSmart) {
        super();
        this.useSmart = useSmart;
    }
    // 重写最新版本的 createComponents；重载 Analyzer 接口，构造分词组件
    @Override
    protected TokenStreamComponents createComponents(String
             fieldName) {
        Tokenizer _IKTokenizer = new IKTokenizer6x(this.useSmart());
        return new TokenStreamComponents(_IKTokenizer);
    }
}
```

实例化 IKAnalyzer6x 即可使用 IK 分词器了，创建默认细粒度切分算法的 IK Analyzer：

```
Analyzer analyzer = new IKAnalyzer6x();
```

创建智能切分算法的 IK Analyzer：

```
Analyzer analyzer = new IKAnalyzer6x(true);
```

2.3.4　中文分词器对比

分词效果会直接影响到文档搜索的准确性，我们对比 Lucene 6.0 中自带的中文智能分词器 SmartChineseAnalyzer 和 IK Analyzer，比较一下哪一个分词器的准确率更高，见代码清单 2-5。

代码清单 2-5　中文分词效果对比

```
import java.io.IOException;
import java.io.StringReader;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.cn.smart.SmartChineseAnalyzer;
import org.apache.lucene.analysis.tokenattributes
       .CharTermAttribute;
import tup.lucene.ik.IKAnalyzer6x;
public class IkVSSmartcn {
private static String str1 = "公路局正在治理解放大道路面积水问题。";
private static String str2 = "IKAnalyzer 是一个开源的，基于 java 语言开发
                             的轻量级的中文分词工具包。";
public static void main(String[] args) throws IOException {
        Analyzer analyzer = null;
        System.out.println("句子一: "+str1);
        System.out.println("SmartChineseAnalyzer 分词结果: ");
        analyzer = new SmartChineseAnalyzer();
        printAnalyzer(analyzer, str1);
        System.out.println("IKAnalyzer 分词结果: ");
        analyzer = new IKAnalyzer6x(true);
        printAnalyzer(analyzer, str1);
        System.out.println("-------------------------------------");
        System.out.println("句子二:"+str2);
        System.out.println("SmartChineseAnalyzer 分词结果: ");
        analyzer = new SmartChineseAnalyzer();
        printAnalyzer(analyzer, str2);
        System.out.println("IKAnalyzer 分词结果: ");
        analyzer = new IKAnalyzer6x(true);
        printAnalyzer(analyzer, str2);
        analyzer.close();
    }
    public static void printAnalyzer(Analyzer analyzer, String str)
       throws IOException {
        StringReader reader = new StringReader(str);
        TokenStream toStream = analyzer.tokenStream(str, reader);
        toStream.reset();       // 清空流
        CharTermAttribute teAttribute = toStream.getAttribute(
                CharTermAttribute.class);
        while (toStream.incrementToken()) {
            System.out.print(teAttribute.toString() + "|");
```

```
            }
            System.out.println();
        }
}
```

运行结果如下：

```
句子一：公路局正在治理解放大道路面积水问题。
SmartChineseAnalyzer 分词结果：
    公路局|正|在|治理|解放|大|道路|面积|水|问题|
IKAnalyzer 分词结果：
    公路局|正在|治理|解放|大道|路面|积水|问题|
-------------------------------------------
句子二：IKAnalyzer 是一个开源的，基于 java 语言开发的轻量级的中文分词工具包。
SmartChineseAnalyzer 分词结果：
ikanalyz|是|一个|开|源|的|基于|java|语言|开发|的|轻量级|的|中文|分词|工具包|
IKAnalyzer 分词结果：
ikanalyzer|是|一个|开源|的|基于|java|语言|开发|的|轻量级|的|中文|分词|工具包
```

从分词结果中可以看到，对于句子一，SmartChineseAnalyzer 没有把“正在”分成一个词，“大道路面积水”分成了“大”“道路”“面积”“水”，而 IK Analyzer 把“正在”分成一个词，把“大道路面积水”分成了“大道”“路面”“积水”。对于句子二，两者的分词效果差不多，但是 SmartChineseAnalyzer 把“开源”分开了。总体而言，IK Analyzer 的中文分词的准确性比 SmartChineseAnalyzer 要高一些。我们在以后的案例和项目中也会选择 IK Analyzer 作为我们的中文分词器。

开源的中分词工具有很多，比如 ICTCLAS 中文分词、Paoding 分词、jcseg 分词等，这里只对比了 Lucene 自带的中文智能分词器和 IK 分词器，读者如果有兴趣可以继续研究。

2.3.5　扩展停用词词典

IK Analyzer 默认的停用词词典为 IKAnalyzer2012_u6/stopword.dic，这个停用词词典并不完整，只有 30 多个英文停用词，推荐使用扩展的停用词词表（下载地址：https://github.com/cseryp/stopwords）。在工程中新增文件 ext_stopword.dic，文件和 IKAnalyzer.cfg.xml 在同一目录，编辑 IKAnalyzer.cfg.xml 把新增的停用词字典写入配置文件，多个停用词字典用逗号隔开，配置如下：

```
<entry key="ext_stopwords">stopword.dic；ext_stopword.dic</entry>
```

2.3.6　扩展自定义词典

IK Analyzer 也支持自定义词典，在 IKAnalyzer.cfg.xml 同一目录新建 ext.dic，把新的词语按行写入文件，然后编辑 IKAnalyzer.cfg.xml，把新增的停用词字典写入配置文件，多个字典用空格隔开，配置如下：

```
<entry key="ext_dict">ext.dic；</entry>
```

比如，对于网络流行语“厉害了我的哥”，默认的词库中没有这个词，要在自定义词典中将其写入以后才能分成一个词。测试自定义词典的代码见代码清单 2-6。

代码清单 2-6　自定义词典测试

```
import java.io.IOException;
import java.io.StringReader;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.tokenattributes
      .CharTermAttribute;
import tup.lucene.ik.IKAnalyzer6x;
public class ExtDicTest {
    private static String str = "厉害了我的哥!中国环保部门即将发布治理北京
                                   雾霾的方法!";
    public static void main(String[] args) throws IOException {
        Analyzer analyzer = new IKAnalyzer6x(true);
        StringReader reader = new StringReader(str);
        TokenStream toStream = analyzer.tokenStream(str, reader);
        toStream.reset();
        CharTermAttribute  teAttribute= toStream.getAttribute(
                CharTermAttribute.class);
        System.out.println("分词结果：");
        while (toStream.incrementToken()) {
            System.out.print(teAttribute.toString() + "|");
        }
        System.out.println("\n");
        analyzer.close();
    }
}
```

运行结果：

```
加载扩展词典：ext.dic
加载扩展停止词典：stopword.dic
分词结果：
厉|害了|的哥|中国|环保部门|发布|治理|北京|雾|霾|方法|
在 ext.dic 中添加自定义词项：
中国环保部门
北京雾霾
厉害了我的哥
再次运行，结果如下：
加载扩展词典：ext.dic
加载扩展停止词典：stopword.dic
```

分词结果：
厉害了我的哥 | 中国环保部门 | 发布 | 治理 | 北京雾霾 | 方法 |

2.4 Lucene 索引详解

介绍完 Lucene 的分词器，我们接着介绍 Lucene 是如何索引文档的，索引文档就是把文档变成索引这种数据结构的过程。

2.4.1 Lucene 字段类型

文档是 Lucene 索引的基本单位，比文档更小的单位是字段，字段是文档的一部分，每个字段由 3 部分组成：名称（name）、类型（type）和取值（value）。字段的取值一般为文本类型（字符串、字符流等）、二进制类型和数值类型。Lucene 中的字段类型主要有以下几种：

- TextField
  TextField 会把该字段的内容索引并词条化，但是不保存词向量。比如，包含整篇文档内容的 body 字段，常常使用 TextField 类型进行索引。
- StringField
  StringField 只会对该字段的内容索引，但是并不词条化，也不保存词向量。字符串的值会被索引为一个单独的词项。比如，有个字段是国家名称，字段名为“country”，以国家“阿尔吉利亚”为例，只索引不词条化是最合适的。
- IntPoint
  IntPoint 适合索引值为 int 类型的字段。IntPoint 是为了快速过滤的，如果需要展示出来需要另存一个字段。比如，商品的数量用字段“productCount”存储，根据商品数量进行过滤操作时可以直接通过 productCount 字段获取结果，但是要想展示商品数量，需要另外再存储一个字段。
- LongPoint
  用法和 IntPoint 类似，区别在 LongPoint 适合索引值为长整型 long 类型的字段。
- FloatPoint
  用法和 IntPoint 类似，区别在 FloatPoint 适合索引值为 float 类型的字段。
- DoublePoint
  用法和 IntPoint 类似，区别在 DoublePoint 适合索引值为 double 类型的字段。
- SortedDocValuesField
  存储值为文本内容的 DocValue 字段，SortedDocValuesField 适合索引字段值为文本内容并且需要按值进行排序的字段。
- SortedSetDocValuesField
  存储多值域的 DocValues 字段，SortedSetDocValuesField 适合索引字段值为文本内容并且需要按值进行分组、聚合等操作的字段。

- NumericDocValuesField
  存储单个数值类型的 DocValues 字段，主要包括（int，long，float，double）。
- SortedNumericDocValuesField
  存储数值类型的有序数组列表的 DocValues 字段。
- StoredField
  StoredField 适合索引只需要保存字段值不进行其他操作的字段。

注：DocValues 是 Lucene 4.X 版本以后新增的重要特性，我们都知道，Lucene 是使用经典的倒排索引的模式来达到快速检索的目的，简单地说，就是建立词项和文档 id 的关系映射，在搜索时，通过类似 hash 算法来快速定位到一个搜索关键词，然后读取文档 id 集合，这样搜索数据是非常高效快速的，当然它也存在一定的缺陷。假如我们需要对数据做一些聚合操作，例如排序、分组时，Lucene 内部会遍历提取所有出现在文档集合的排序字段，然后再次构建一个最终的排好序的文档集合，这个过程全部维持在内存中操作，而且如果排序数据量巨大的话，非常容易造成内存溢出和性能缓慢。基于此，在 lucene 4.X 之后出现了 DocValues 这一新特性，DocValues 其实是 Lucene 在构建索引时额外建立一个有序的基于 document=>field/value 的映射列表。在构建索引时会对开启 docvalues 的字段额外构建一个已经排好序的文档到字段级别的一个列式存储映射，它减轻了在排序和分组时对内存的依赖，而且大大提升了这个过程的性能，当然它也会耗费一定的磁盘空间。

2.4.2　索引文档示例

首先新建一个代表新闻的实体类 News.java，为了简单起见，我们给新闻对象设置新闻 id、新闻标题、新闻内容和评论数 4 个属性，然后提供相应的构造方法、Setter 和 Getter 方法，代码见代码清单 2-7。

代码清单 2-7　News.java

```
public class News {
    private int id;
    private String title;
    private String content;
    private int reply;
    public News() {
    }
    public News(int id, String title, String content, int reply) {
        super();
        this.id = id;
        this.title = title;
        this.content = content;
        this.reply = reply;
    }
    public int getId() {
```

```
            return id;
        }
        public void setId(int id) {
            this.id = id;
        }
        public String getTitle() {
            return title;
        }
        public void setTitle(String title) {
            this.title = title;
        }
        public String getContent() {
            return content;
        }
        public void setContent(String content) {
            this.content = content;
        }
        public int getReply() {
            return reply;
        }
        public void setReply(int reply) {
            this.reply = reply;
        }
    }
```

Lucene 索引文档要依靠一个 IndexWriter 对象，创建 IndexWriter 需要提供两个参数，一个是 IndexWriterConfig 对象，该对象可以设置创建索引使用哪种分词器，另一个是索引的保存路径。IndexWriter 对象的 addDocument()方法用于添加文档，该方法的参数为 Document 对象。IndexWriter 对象一次可以添加多个文档，最后调用 commit()方法生成索引。CreateIndex.java 是我们创建索引的代码，我们首先创建了 3 个 News 对象，每条新闻添加上新闻 ID、新闻标题、新闻内容和新闻的评论条数，后面的代码会把这 3 个 News 对象写入 Lucene 索引。IKAnalyzer 对象用于指定创建索引时的分词器，它作为参数传入 IndexWriterConfig()方法中实例化一个 IndexWriterConfig 对象。IndexWriterConfig 对象的 setOpenMode()方法可以设置索引的打开方式，传入 OpenMode.CREATE 参数表示先清空索引再重新创建，传入 CREATE_OR_APPEND 参数表示如果索引不存在会新建，已存在则附加。Directory 对象用于表示索引的位置，把索引路径和 IndexWriterConfig 对象传入 IndexWriter()方法，实例化 IndexWriter 对象，之后就可以通过 IndexWriter 对象进行文档的操作。

文档是 Lucene 索引和搜索的基本单位，在代码中 Document 类表示文档，比文档更小的单位是域，也可以称为字段，一个文档可以有多个域。FieldType 对象用于指定域的索引信息，例如是否解析、是否存储、是否保存词项频率、位移信息等。FieldType 对象的 setIndexOptions()方法可以设定域的索引选项，可选参数及含义如下：

- IndexOptions.DOCS
  只索引文档，词项频率和位移信息不保存。
- IndexOptions.DOCS_AND_FREQS
  只索引文档和词项频率，位移信息不保存。
- IndexOptions.DOCS_AND_FREQS_AND_POSITIONS
  索引文档、词项频率和位移信息。
- IndexOptions.DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS
  索引文档、词项频率、位移信息和偏移量。
- IndexOptions.NONE
  不索引。

全文搜索中很重要的一项需求是关键字的高亮，要想准确地获取位置信息以及一些偏移量就需要在创建索引的时候进行记录，如果信息不准确，那么可能在搜索的时候就会发生错位，反映到网页上就是标注了不该标注的字，没有标注该标的内容。在索引的时候，可以使用 FieldType 对象提供的方法设置相对增量和位移信息。

- setStored（boolean value）
  参数默认值为 false，设置为 true 存储字段值。
- setTokenized（boolean value）
  参数设置为 true，会使用配置的分词器对字段值进行词条化。
- setStoreTermVectors（boolean value）
  参数为 true，保存词向量。
- setStoreTermVectorPositions（boolean value）
  参数为 true，保存词项在词向量中的位移信息。
- setStoreTermVectorOffsets（boolean value）
  参数为 true，保存词项在词向量中的偏移信息。

代码清单 2-8　CreateIndex.java

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Date;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.document.FieldType;
import org.apache.lucene.document.IntPoint;
import org.apache.lucene.document.SortedNumericDocValuesField;
import org.apache.lucene.document.StoredField;
import org.apache.lucene.index.IndexOptions;
```

```
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.index.IndexWriterConfig.OpenMode;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import tup.lucene.ik.IKAnalyzer6x;
public class CreateIndex {
    public static void main(String[] args) {
        // 创建 3 个 News 对象
        News news1 = new News();
        news1.setId(1);
        news1.setTitle("习近平会见美国总统奥巴马，学习国外经验");
        news1.setContent("国家主席习近平 9 月 3 日在杭州西湖国宾馆会见前来出席
                二十国集团领导人杭州峰会的美国总统奥巴马...");
        news1.setReply(672);
        News news2 = new News();
        news2.setId(2);
        news2.setTitle("北大迎 4380 名新生 农村学生 700 多人近年最多");
        news2.setContent("昨天，北京大学迎来 4380 名来自全国各地及数十个国家
                的本科新生。其中，农村学生共 700 余名，为近年最多...");
        news2.setReply(995);

        News news3 = new News();
        news3.setId(3);
        news3.setTitle("特朗普宣誓(Donald Trump)就任美国第 45 任总统");
        news3.setContent("当地时间 1 月 20 日，唐纳德·特朗普在美国国会宣誓就
                职，正式成为美国第 45 任总统。");
        news3.setReply(1872);

        // 创建 IK 分词器
        Analyzer analyzer = new IKAnalyzer6x();
        IndexWriterConfig icw = new IndexWriterConfig(analyzer);
        icw.setOpenMode(OpenMode.CREATE);
        Directory dir = null;
        IndexWriter inWriter = null;
        // 索引目录
        Path indexPath = Paths.get("indexdir");
        // 开始时间
        Date start = new Date();
        try {
            if (!Files.isReadable(indexPath)) {
              System.out.println("Document directory '" + indexPath.
                toAbsolutePath()+ "' does not exist or is not
               readable, please check the path");
```

```
  System.exit(1);
}
dir = FSDirectory.open(indexPath);
inWriter = new IndexWriter(dir, icw);
//设置新闻 ID 索引并存储
FieldType idType = new FieldType();
idType.setIndexOptions(IndexOptions.DOCS);
idType.setStored(true);
//设置新闻标题索引文档、词项频率、位移信息和偏移量，存储并词条化
FieldType titleType = new FieldType();
titleType.setIndexOptions(IndexOptions.
DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS);
titleType.setStored(true);
titleType.setTokenized(true);

FieldType contentType = new FieldType();
contentType.setIndexOptions(IndexOptions.
DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS);
contentType.setStored(true);
contentType.setTokenized(true);
contentType.setStoreTermVectors(true);
contentType.setStoreTermVectorPositions(true);
contentType.setStoreTermVectorOffsets(true);
contentType.setStoreTermVectorPayloads(true);
Document doc1 = new Document();
doc1.add(new Field("id", String.valueOf(news1.getId()),
        idType));
doc1.add(new Field("title", news1.getTitle(),
        titleType));
doc1.add(new Field("content",news1.getContent(),
            contentType));
doc1.add(new IntPoint ("reply", news1.getReply()));
doc1.add(new StoredField("reply_display",
            news1.getReply()));
Document doc2 = new Document();
doc2.add(new Field("id", String.valueOf(news2.getId()),
         idType));
doc2.add(new Field("title", news2.getTitle(),
        titleType));
doc2.add(new Field("content", news2.getContent(),
        contentType));
doc2.add(new IntPoint("reply", news2.getReply()));
doc2.add(new StoredField("reply_display",
```

```
                    news2.getReply()));
            Document doc3 = new Document();
            doc3.add(new Field("id", String.valueOf(news3.getId()),
                    idType));
            doc3.add(new Field("title"news3.getTitle(), titleType));
            doc3.add(new Field("content", news3.getContent(),
                    contentType));
            doc3.add(new IntPoint("reply", news3.getReply()));
            doc3.add(new StoredField("reply_display", news3.
                    getReply()));
            inWriter.addDocument(doc1);
            inWriter.addDocument(doc2);
            inWriter.addDocument(doc3);
            inWriter.commit();
            inWriter.close();
            dir.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        Date end = new Date();
        System.out.println("索引文档用时:" + (end.getTime() - start.
            getTime()) + " milliseconds");
    }
}
```

运行结果：

```
**********开始创建索引**********
加载扩展词典：ext.dic
加载扩展停止词典：stopword.dic
加载扩展停止词典：ext_stopword.dic
索引文档用时:884 milliseconds
**********索引创建完成**********
```

刷新工程，在 indexdir 目录下可以看到生成的索引文件：

```
_0.cfe      _0.cfs      _0.si      segments_1      write.lock
```

2.4.3　Luke 中查看索引

索引创建完成以后生成了一批特殊格式的文件，可以使用索引查看工具 Luke 来查看。启动 Luke，Path 选为 Lucene 索引的位置，在本例中为 LuceneDemo/indexdir 的绝对路径。初次启动 Luke 的界面如图 2-9 所示。

Luke 启动之后，Overview 选项卡界面显示了打开的索引的基本信息：索引路径、字段数、文档数、词项数、索引版本、词项频率等，如图 2-10 所示。

切换到 Documents 选项卡即可查询写入的文档，Luke 查看索引结果如图 2-11 所示。可以

看到 reply 字段为空，是因为 IntPoint 字段类型是为了快速过滤的，如果需要展示出来需要另存一个字段，我们用 reply_display 来存储。

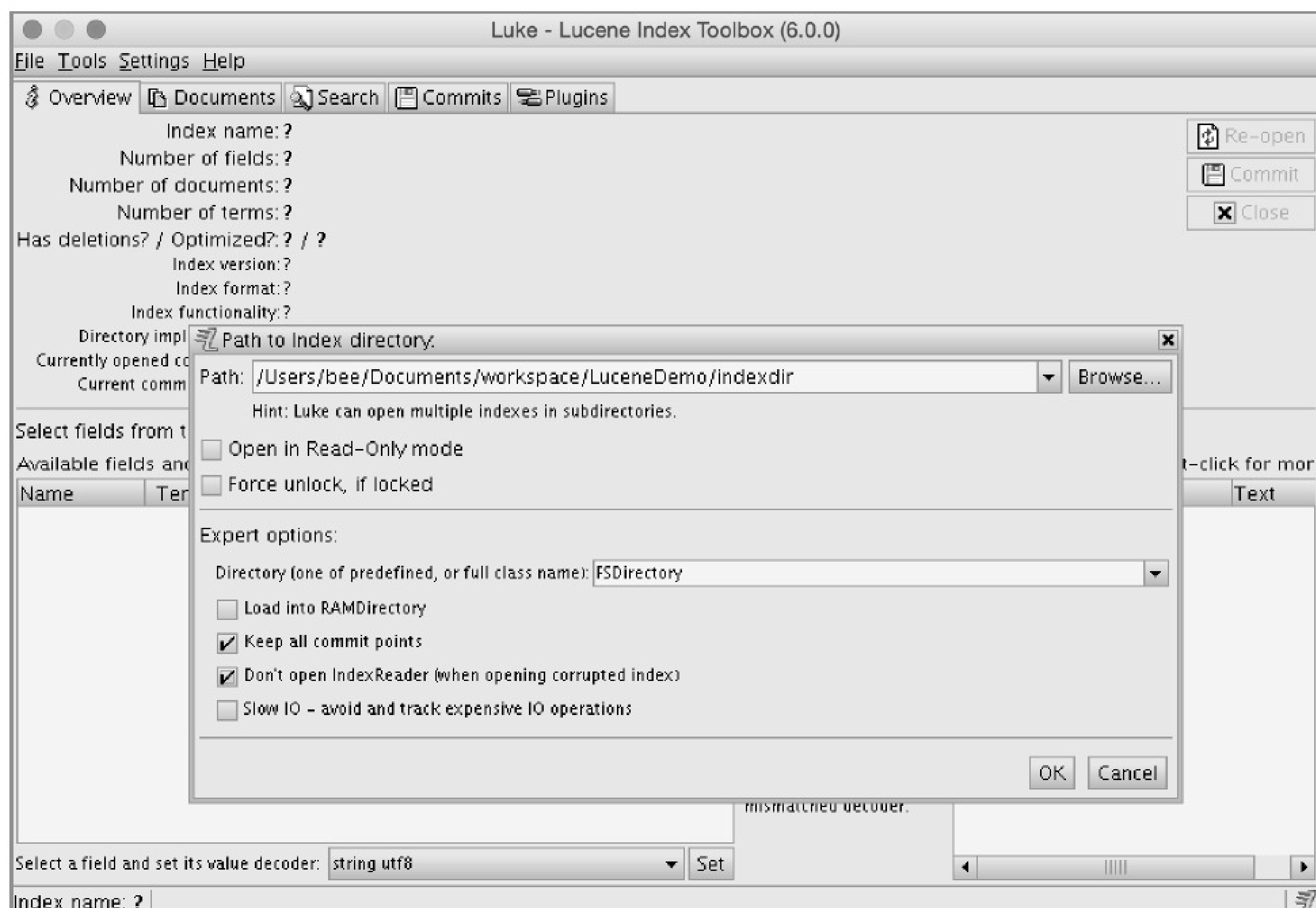


图 2-9　Luke 中打开索引路径

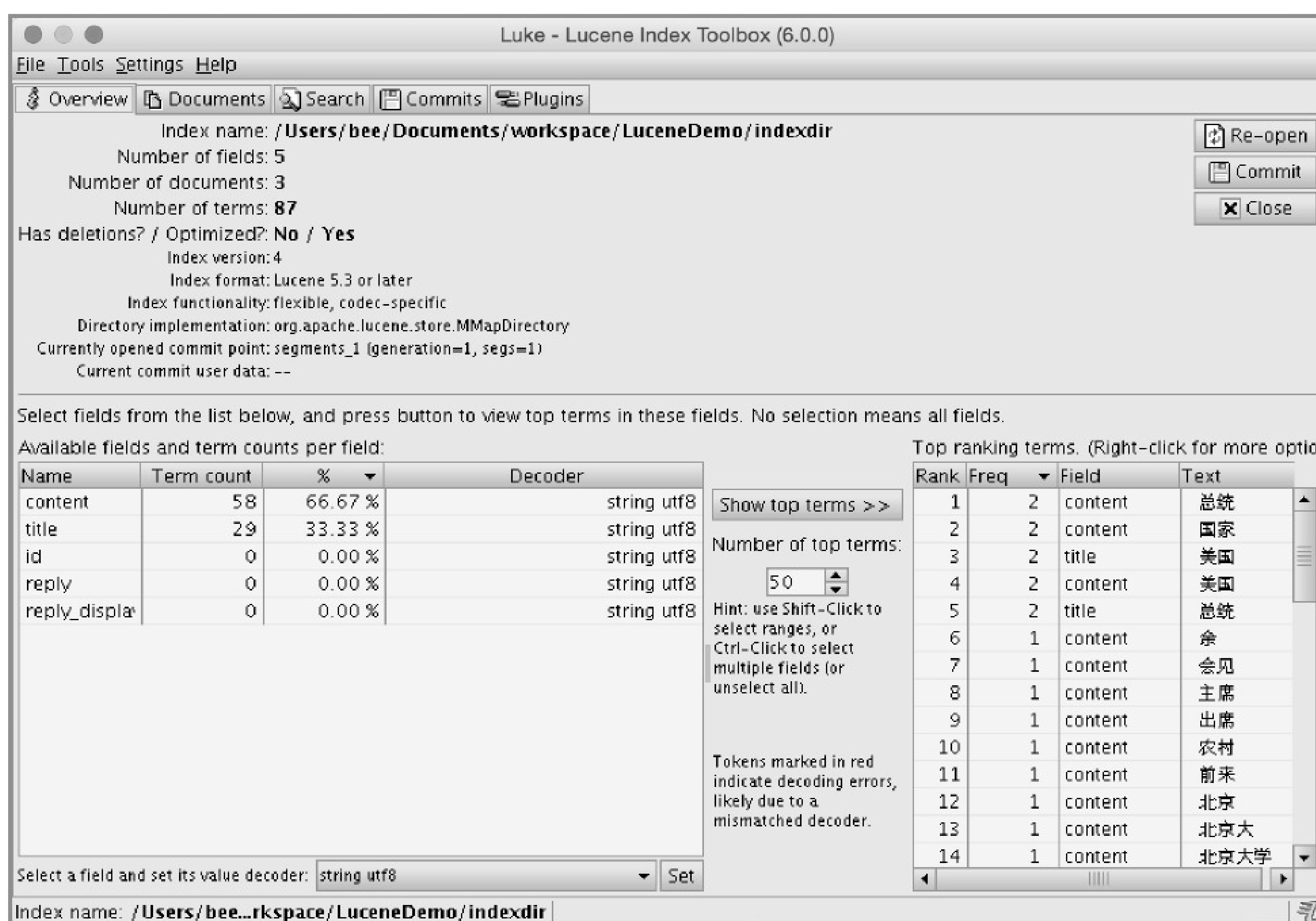


图 2-10　Luke 中查看索引

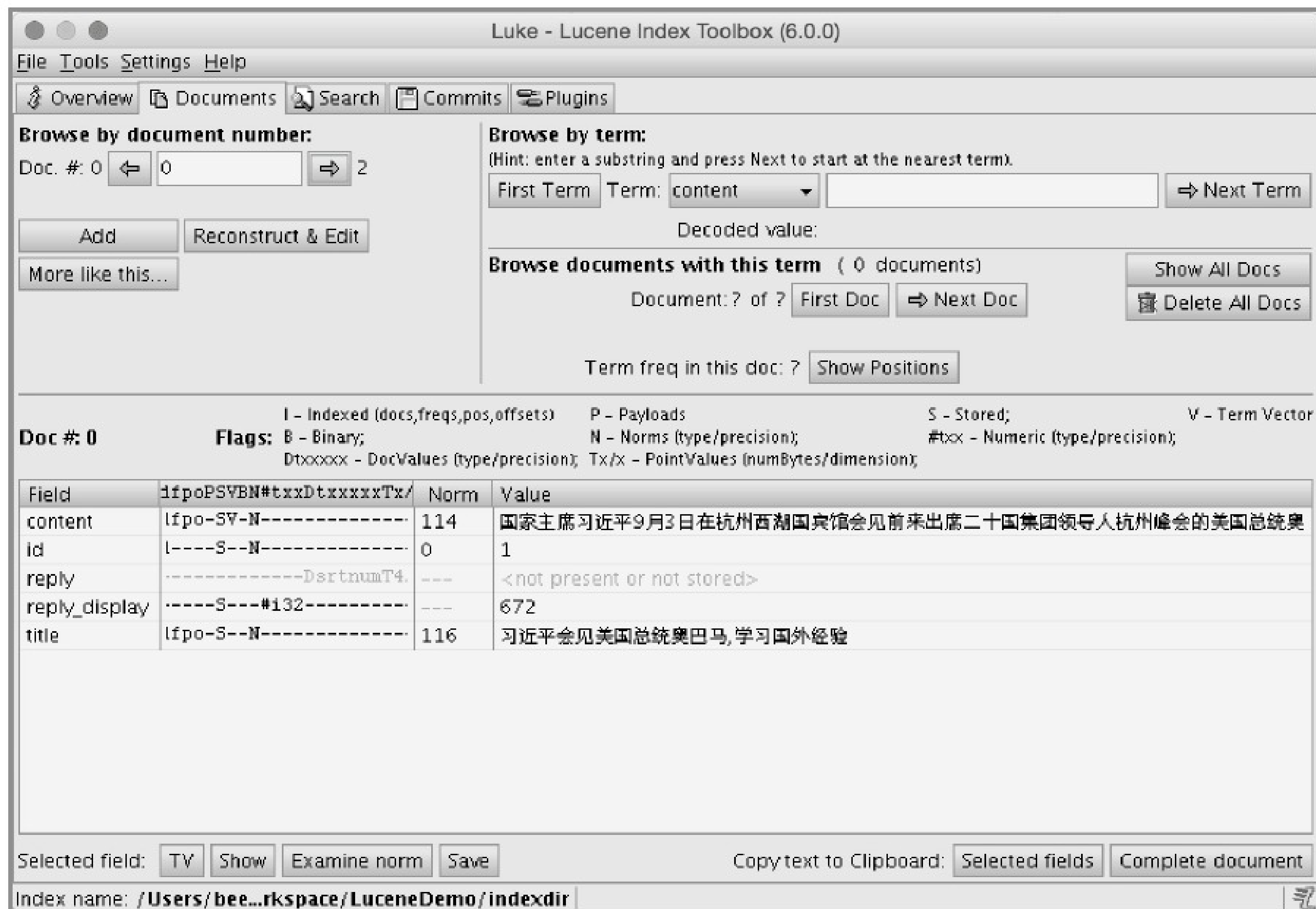


图 2-11 Luke 中查看索引

2.4.4 索引的删除

上一小节介绍了 Lucene 是如何索引文档的，索引同样存在 CRUD 操作，这一小节通过示例介绍索引的删除和更新操作。删除与更新和新增一样，也是通过 IndexWriter 对象来操作的，IndexWrite 对象的 deleteDocuments()方法用于实现索引的删除，updateDocument()方法用于实现索引的更新。删除索引的代码见代码清单 2-9，该示例实现了根据 Term 来删除单个或多个 Document，删除 title 中包含关键词“美国“的文档。

代码清单 2-9

```
import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.index.Term;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import tup.lucene.ik.IKAnalyzer6x;
public class DeleteIndex {
    public static void main(String[] args) {
        // 删除 title 中含有关键词“美国”的文档
        deleteDoc("title", "美国");
```

```
    }
    public static void deleteDoc(String field, String key) {
        Analyzer analyzer = new IKAnalyzer6x();
        IndexWriterConfig icw = new IndexWriterConfig(analyzer);
        Path indexPath = Paths.get("indexdir");
        Directory directory;
        try {
            directory = FSDirectory.open(indexPath);
            IndexWriter indexWriter = new IndexWriter(directory,
                                      icw);
            indexWriter.deleteDocuments(new Term(field, key));
            indexWriter.commit();
            indexWriter.close();
            System.out.println("删除完成！");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

除此之外，IndexWriter 还提供了以下方法：

- DeleteDocuments(Query query)：根据 Query 条件来删除单个或多个 Document。
- DeleteDocuments(Query[] queries)：根据 Query 条件来删除单个或多个 Document。
- DeleteDocuments(Term term)：根据 Term 来删除单个或多个 Document。
- DeleteDocuments(Term[] terms)：根据 Term 来删除单个或多个 Document。
- DeleteAll(): 删除所有的 Document。

使用 IndexWriter 进行 Document 删除操作时，文档并不会立即被删除，而是把这个删除动作缓存起来，当 IndexWriter.Commit()或 IndexWriter.Close()时，删除操作才会被真正执行。

2.4.5　索引的更新

索引更新操作实质上是先删除索引，再重新建立新的文档，见代码清单 2-10。

代码清单 2-10

```
import java.nio.file.Path;
import java.nio.file.Paths;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field.Store;
import org.apache.lucene.document.TextField;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
```

```
import org.apache.lucene.index.Term;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import tup.lucene.ik.IKAnalyzer6x;
public class UpdateIndex {
  public static void main(String[] args) {
        Analyzer analyzer = new IKAnalyzer6x();
        IndexWriterConfig icw = new IndexWriterConfig(analyzer);
        Path indexPath = Paths.get("indexdir");
        Directory directory;
        try {
            directory = FSDirectory.open(indexPath);
            IndexWriter indexWriter = new IndexWriter(directory, icw);
            Document doc = new Document();
            doc.add(new TextField("id", "2", Store.YES));
            doc.add(new TextField("title", "北京大学开学迎来 4380 名新生
                   ", Store.YES));
            doc.add(new TextField("content", " 昨天，北京大学迎来 4380 名
                  来自全国各地及数十个国家的本科新生。其中，农村学生共 700 余
                  名，为近年最多...", Store.YES));
            indexWriter.updateDocument(new Term("title", "北大"),
                  doc);
            indexWriter.commit();
            indexWriter.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

上面的代码中新建了一个 IndexWriter 对象和 Document 对象，通过 updateDocument()方法完成更新操作。Term 对象用于定位文档，查找 title 中含有"北大"的文档，然后用新的文档替换原文档，这样就完成了索引的更新操作。

2.5　Lucene 查询详解

文档索引完成以后就可以对其进行搜索，当用户输入一个关键字，搜索引擎接收到后，并不是立刻就将它放入后台开始进行关键字的检索，而应当首先对这个关键字进行一定的分析和处理，使之成为一种后台可以理解的形式，只有这样，才能提高检索的效率，同时检索出更加有效的结果。

2.5.1　搜索入门

在 Lucene 中，处理用户输入的查询关键词其实就是构建 Query 对象的过程。Lucene 搜索文档需要实例化一个 IndexSearcher 对象，IndexSearcher 对象的 search()方法完成搜索过程，Query 对象作为 search()方法的对象。搜索结果会保存在一个 TopDocs 类型的文档集合中，遍历 TopDocs 集合输出文档信息。见代码清单 2-11。

代码清单 2-11

```
package tup.lucene.queries;
import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.queryparser.classic.ParseException;
import org.apache.lucene.queryparser.classic.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import tup.lucene.ik.IKAnalyzer6x;
public class QueryParseTest{
public static void main(String[] args)
    throws ParseException, IOException {
        String field = "title";
        Path indexPath = Paths.get("indexdir");
        Directory dir = FSDirectory.open(indexPath);
        IndexReader reader = DirectoryReader.open(dir);
        IndexSearcher searcher = new IndexSearcher(reader);
        Analyzer analyzer = new IKAnalyzer6x();
        QueryParser parser = new QueryParser(field, analyzer);
        parser.setDefaultOperator(Operator.AND);
        Query query = parser.parse("农村学生"); // 查询关键词
        System.out.println("Query:"+query.toString());
        // 返回前 10 条
        TopDocs tds = searcher.search(query, 10);
        for (ScoreDoc sd : tds.scoreDocs) {
            Document doc = searcher.doc(sd.doc);
```

```
                System.out.println("DocID:" + sd.doc);
                System.out.println("id:" + doc.get("id"));
                System.out.println("title:" + doc.get("title"));
                System.out.println("文档评分:" + sd.score);
            }
            dir.close();
            reader.close();
        }
    }
```

注意上面代码中的这几行代码：

```
QueryParser parser = new QueryParser(field, analyzer);
Query query = parser.parse("农村学生");
parser.setDefaultOperator(Operator.AND);
TopDocs topDocs = searcher.search(query, 10);
```

QueryParser 实际上就是一个解析用户输入的工具，可以通过扫描用户输入的字符串生成 Query 对象。当使用 QueryParser 构建用户 Query 时，要搜索的 field 和 analyzer 对象作为参数传入 QueryParser 类，告诉 QueryParser 在哪个字段内查找该关键字信息以及搜索时使用什么样的分词器。这里设置要查询的字段为 title 字段，查询所用的分词器为 IK 智能分词，查询关键词为“农村学生”，关键词经过分词器分成“农村”和“学生”两个词项，title 中含有这两个词项中的任何一个的文档都是本次查询的匹配文档。如果只想返回同时包含两个词项的文档，可以通过 setDefaultOperator()方法把关键词理解为 AND 操作。

注意，在结果中打印了 DocID 和 id，前者是文档 ID，是 Lucene 为索引的每个文档做的标记，后者是文档内部的 id 字段。

运行结果如下：

```
加载扩展词典：ext.dic
加载扩展停止词典：stopword.dic
加载扩展停止词典：ext_stopword.dic
Query:+title:农村 +title:学生
DocID:1
id:2
title:北大迎 4380 名新生 农村学生 700 多人近年最多
文档评分:1.6022166
```

改变上面搜索实例的 Query 对象就可以实现不同种类型的搜索需求，比如多域查询、布尔查询、模糊查询、通配符查询等。

2.5.2 多域搜索（MultiFieldQueryParser）

2.4.1 小节中介绍的 QueryParser 可以搜索单个字段，而 MultiFieldQueryParser 则可以查询多个字段。通过 MultiFieldQueryParser 对象生成 Query 对象的代码如下：

```
String[] fields = { "title", "content" };
```

```
Analyzer analyzer = new IKAnalyzer6x(true);
MultiFieldQueryParser parser = new MultiFieldQueryParser(fields, analyzer);
Query multiFieldQuery = parser.parse("日本");
```

通过 IndexSearcher 搜索文档和打印结果的代码和 2.4.1 中的一样，这里省略，给出查询结果：

```
title:日本 content:日本
DocID:0
id:1
title:安倍晋三本周会晤特朗普 将强调日本对美国益处
content:日本首相安倍晋三计划 2 月 10 日在华盛顿与美国总统特朗普举行会晤时提出加
        大日本在美国投资的设想
文档评分:2.0341508
```

2.5.3　词项搜索（TermQuery）

TermQuery 是最简单也是最常用的 Query。TermQuery 可以理解成为“词项搜索”，在搜索引擎中最基本的搜索就是在索引中搜索某一词条，而 TermQuery 就是用来完成这项工作的。在 Lucene 中词条是最基本的搜索单位，从本质上来讲一个词条其实就是一个 key/value 对。只不过这个 key 是字段名，而 value 则表示字段中所包含的某个关键字。要使用 TermQuery 进行搜索首先需要构造一个 Term 对象，示例代码如下：

```
Term term = new Term("title", "美国");
```

然后使用 Term 对象为参数来构造一个 TermQuery 对象，代码设置如下：

```
Query termQuery = new TermQuery(term);
```

这样所有在“title”字段中包含有“美国”的文档都会在使用 TermQuery 进行查询时作为符合查询条件的结果返回。同样省略 IndexSearcher 搜索文档和打印结果的代码，运行结果如下：

```
Query:title:美国
DocID:2
id:3
title:特朗普宣誓就任美国第 45 任总统
文档评分:0.53872687
DocID:0
id:1
title:安倍晋三本周会晤特朗普 将强调日本对美国益处
文档评分:0.38388318
```

2.5.4　布尔搜索（BooleanQuery）

BooleanQuery 也是实际开发过程中经常使用的一种 Query 查询，它其实是一个组合的 Query，在使用时可以把各种 Query 对象添加进去并标明它们之间的逻辑关系。BooleanQuery 本

身来讲是一个布尔子句的容器，它提供了专门的 API 方法往其中添加子句，并标明它们之间的关系。下面的代码中，创建了两个 TermQuery，BooleanClause 对象可以指定查询的包含关系，并作为参数通过 BooleanQuery.Builder()构造布尔查询。查询 title 字段中包含关键词“美国”并且“content”字段中不包含日本的文档，代码如下：

```
Query query1 = new TermQuery(new Term("title", "美国"));
Query query2 = new TermQuery(new Term("content", "日本"));
BooleanClause bc1=new BooleanClause(query1，Occur.MUST);
BooleanClause bc2=new BooleanClause(query2，Occur.MUST_NOT);
BooleanQuery boolQuery=new BooleanQuery.Builder()
    .add(bc1).add(bc2).build();
```

查询结果如下：

```
Query:+title:美国 -content:日本
DocID:2
id:1
title:特朗普宣誓就任美国第 45 任总统
content:当地时间 1 月 20 日，唐纳德·特朗普在美国国会宣誓就职，正式成为美国第 45任总统。
文档评分:0.53872687
```

2.5.5　范围搜索（RangeQuery）

有时用户需要查找满足一定范围的文档，比如查找某一时间段内的所有文档，Lucene 提供了 RangeQuery 查询来满足这种需求。

RangeQuery 表示在某范围内的搜索条件，实现从一个开始词条到一个结束词条的搜索功能，在查询时“开始词条”和“结束词条”可以包含在内也可以不被包含在内。查询新闻回复条数在 500 条到 1000 条之间的有哪些，构成 Query 对象的代码如下：

```
Query rangeQuery=IntPoint.newRangeQuery("reply", 500, 1000);
```

同样省略 IndexSearcher 搜索文档和打印结果的代码，查询结果如下：

```
Query:reply:[500 TO 1000]
DocID:0
id:1
title:安倍晋三本周会晤特朗普 将强调日本对美国益处
Reply:672
文档评分:1.0
DocID:1
id:2
title:北大迎 4380 名新生 农村学生 700 多人近年最多
Reply:995
文档评分:1.0
```

2.5.6　前缀搜索（PrefixQuery）

PrefixQuery 就是使用前缀来进行查找的。通常情况下，首先定义一个词条 Term。该词条包含要查找的字段名以及关键字的前缀，然后通过该词条构造一个 PrefixQuery 对象，就可以进行前缀查找了。查询包含以“学”开头的词项的文档，构造 Query 对象的代码如下：

```
Term term = new Term("title", "学");
Query prefixQuery = new PrefixQuery(term);
```

同样省略 IndexSearcher 搜索文档和打印结果的代码，查询结果如下：

```
Query:title:学*
DocID:1
id:2
title:北大迎 4380 名新生 农村学生 700 多人近年最多
文档评分:1.0
```

2.5.7　多关键字搜索（PhraseQuery）

除了普通的 TermQuery 外，Lucene 还提供了一种 Phrase 查询功能。用户在搜索引擎中进行搜索时，常常查找的并非是一个简单的单词，很有可能是几个不同的关键字。这些关键字之间要么是紧密相连的，成为一个精确的短语，要么在这几个关键字之间还插有其他无关的内容。

PhraseQuery 正是 Lucene 所提供的满足上述需求的一种 Query 对象。它的 add 方法可以让用户向其内部添加关键字，在添加完毕后，用户还可以通过 setSlop()方法来设定一个称之为“坡度”的变量来确定关键字之间是否允许或允许多少个无关词汇的存在。

```
PhraseQuery.Builder builder = new PhraseQuery.Builder();
builder.add(new Term("title", "日本"), 2);
builder.add(new Term("title", "美国"), 3);
PhraseQuery phraseQuery = builder.build();
打印 phraseQuery 对象:
Query:title:"? ? 日本 美国"
```

2.5.8　模糊搜索（FuzzyQuery）

FuzzyQuery 是一种模糊查询，它可以简单地识别两个相近的词语。例如，由于拼写错误把“Trump”拼成“Trmp”或者“Tramp”，使用 FuzzyQuery 仍可搜索到正确的结果，代码如下：

```
Term term = new Term("title", "Tramp");
FuzzyQuery fuzzyQuery = new FuzzyQuery(term);
Query:title:Tramp~2
DocID:2
id:3
```

```
title:特朗普宣誓(Donald Trump)就任美国第 45 任总统
文档评分:0.6056149
```

2.5.9 通配符搜索（WildcardQuery）

Lucene 也提供了通配符的查询，就是使用 WildcardQuery。示例如下：

```
WildcardQuery wildcardQuery=new WildcardQuery(new Term(field, "学?"));
Query:title:学*
DocID:1
id:2
title:北大迎 4380 名新生 农村学生 700 多人近年最多
文档评分:1.0
```

2.6 Lucene 查询高亮

高亮功能一直都是全文检索的一项非常优秀的模块，在一个标准的搜索引擎中，高亮的返回命中结果，几乎是必不可少的一项需求，因为通过高亮，可以在搜索界面上快速标记出用户的检索关键词，从而减少了用户自己寻找想要的结果的时间，在一定程度上大大提高了用户的体验性和友好度。Lucene 查询高亮的例子见代码清单 2-12。

代码清单 2-12

```
import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.document.Document;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.queryparser.classic.ParseException;
import org.apache.lucene.queryparser.classic.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.search.highlight.Fragmenter;
import org.apache.lucene.search.highlight.Highlighter;
import org.apache.lucene.search.highlight.InvalidTokenOffsetsException;
import org.apache.lucene.search.highlight.QueryScorer;
import org.apache.lucene.search.highlight.SimpleHTMLFormatter;
```

```
import org.apache.lucene.search.highlight.SimpleSpanFragmenter;
import org.apache.lucene.search.highlight.TokenSources;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import tup.lucene.ik.IKAnalyzer6x;
public class HighlighterTest {
    public static void main(String[] args) throws IOException,
InvalidTokenOffsetsException, ParseException {
        String field = "title";
        Path indexPath = Paths.get("indexdir");
        Directory dir = FSDirectory.open(indexPath);
        IndexReader reader = DirectoryReader.open(dir);
        IndexSearcher searcher = new IndexSearcher(reader);
        Analyzer analyzer = new IKAnalyzer6x();
        QueryParser parser = new QueryParser(field, analyzer);
        Query query = parser.parse("北大");
        System.out.println("Query:" + query);
        QueryScorer score = new QueryScorer(query, field);
        SimpleHTMLFormatter fors = new SimpleHTMLFormatter("<span
                style=\"color:red; \">", "</span>");        //定制高亮标签
        Highlighter highlighter = new Highlighter(fors, score);
        // 高亮分词器
        TopDocs tds = searcher.search(query, 10);
        for (ScoreDoc sd : tds.scoreDocs) {
            Document doc = searcher.doc(sd.doc);
            System.out.println("id:" + doc.get("id"));
            System.out.println("title:" + doc.get("title"));
            TokenStream tokenStream = TokenSources.getAnyTokenStream
                (searcher. getIndexReader(), sd.doc, field, analyzer);
            // 获取 tokenstream
            Fragmenter fragment = new SimpleSpanFragmenter(score);
            highlighter.setTextFragmenter(fragment);
            String str = highlighter.getBestFragment(tokenStream,
                          doc.get(field));        //获取高亮的片段
            System.out.println("高亮的片段:" + str);
        }
        dir.close();
        reader.close();
    }
}
```

运行结果：

```
加载扩展词典：ext.dic
加载扩展停止词典：stopword.dic
加载扩展停止词典：ext_stopword.dic
Query:title:北大
id:2
title:北大迎 4380 名新生 农村学生 700 多人近年最多
高亮的片段:<span style="color:red;">北大</span>迎 4380 名新生 农村学生 700 多人近
年最多
```

2.7　Lucene 新闻高频词提取

2.7.1　问题提出

给出一篇新闻文档，统计出现频率最高的有哪些词语。

2.7.2　需求分析

关于文本关键词提取的算法有很多，开源工具也不止一种。这里只介绍如何从 Lucene 索引中提取词项频率的 Top N。索引过程的本质是一个词条化的生存倒排索引的过程，词条化会从文本中去除标点符号、停用词等，最后生成词项。在代码中实现的思路是使用 IndexReader 的 getTermVector 获取文档的某一个字段的 Terms，从 terms 中获取 tf（term frequency），拿到词项的 tf 以后放到 map 中降序排序，取出 Top-N。

2.7.3　编程实现

在百度新闻上随机找了一篇新闻《李开复：无人驾驶进入黄金时代 AI 有巨大投资机会》，新闻内容为李开复关于人工智能的主题演讲。把新闻的文本内容放在 testfile/news.txt 文件中，索引文档代码见代码清单 2-13。

代码清单 2-13

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.nio.file.Paths;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.document.FieldType;
import org.apache.lucene.index.IndexOptions;
```

```
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.index.IndexWriterConfig.OpenMode;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
import lucene.ik.IKAnalyzer6x;
public class IndexDocs {
    public static void main(String[] args) throws IOException {
        File newsfile = new File("testfile/lucene.txt");
        String text1 = textToString(newsfile);
        Analyzer smcAnalyzer = new IKAnalyzer6x(true);
        IndexWriterConfig indexWriterConfig = new IndexWriterConfig(
                smcAnalyzer);
        indexWriterConfig.setOpenMode(OpenMode.CREATE);
        // 索引的存储路径
        Directory directory = null;
        // 索引的增删改由 indexWriter 创建
        IndexWriter indexWriter = null;
        directory = FSDirectory.open(Paths.get("indexdir"));
        indexWriter = new IndexWriter(directory, indexWriterConfig);
        // 新建 FieldType，用于指定字段索引时的信息
        FieldType type = new FieldType();
        // 索引时保存文档、词项频率、位置信息、偏移信息
        type.setIndexOptions(IndexOptions.DOCS_AND_FREQS_AND_
                            POSITIONS_AND_ OFFSETS);
        type.setStored(true); // 原始字符串全部被保存在索引中
        type.setStoreTermVectors(true); // 存储词项量
        type.setTokenized(true); // 词条化
        Document doc1 = new Document();
        Field field1 = new Field("content", text1, type);
       doc1.add(field1);
       indexWriter.addDocument(doc1);
       indexWriter.close();
       directory.close();
}
public static String textToString(File file) {
    StringBuilder result = new StringBuilder();
    try {
    // 构造一个 BufferedReader 类来读取文件
    BufferedReader br = new BufferedReader(new FileReader(file));
    String str = null;
   // 使用 readLine 方法，一次读一行
    while ((str = br.readLine()) != null) {
```

```
            result.append(System.lineSeparator() + str);
        }
            br.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result.toString();
    }
    }
```

获取新闻热词的代码见代码清单 2-14。

代码清单 2-14

```
    import java.io.IOException;
    import java.nio.file.Paths;
    import java.util.ArrayList;
    import java.util.Collections;
    import java.util.Comparator;
    import java.util.HashMap;
    import java.util.List;
    import java.util.Map;
    import java.util.Map.Entry;
    import org.apache.lucene.index.DirectoryReader;
    import org.apache.lucene.index.IndexReader;
    import org.apache.lucene.index.Terms;
    import org.apache.lucene.index.TermsEnum;
    import org.apache.lucene.store.Directory;
    import org.apache.lucene.store.FSDirectory;
    import org.apache.lucene.util.BytesRef;
    public class GetTopTerms {
    public static void main(String[] args) throws IOException {
    Directory directory =FSDirectory. open(Paths. get("indexdir"));
    IndexReader reader = DirectoryReader.open(directory);
       //因为只索引了一个文档，所以 DocID 为 0
       //通过 getTermVector 获取 content 字段的词项
     Terms terms = reader.getTermVector(0,  "content");
            // 遍历词项
     TermsEnum termsEnum = terms.iterator();
     Map<String,  Integer> map = new HashMap<String,  Integer>();
            BytesRef thisTerm;
            while ((thisTerm=termsEnum.next())!= null) {
                String termText = thisTerm.utf8ToString(); // 词项
                // 通过 totalTermFreq()方法获取词项频率
                map.put(termText,  (int) termsEnum.totalTermFreq());
```

```
            }
            // 按 value 排序
            List<Map.Entry<String, Integer>> sortedMap = new
            ArrayList<Map.Entry<String, Integer>>(map.entrySet());
            Collections.sort(sortedMap, new Comparator<Map.Entry<
                String, Integer>>() {
                public int compare(Map.Entry<String, Integer> o1,
                Map. Entry<String, Integer> o2)   {
                    return (o2.getValue() - o1.getValue());
                }
            });
            getTopN(sortedMap, 10);
        }
        // 获取 top-n
        public static void getTopN(List<Entry<String, Integer>>
                            sortedMap, int N) {
            for (int i = 0; i < N; i++) {
                System.out.println(sortedMap.get(i).getKey() + ":" +
                            sortedMap.get(i).getValue());
            }
        }
    }
```

运行结果：

```
人工智能:61
领域:34
公司:23
无人驾驶:20
投资:20
互联网:18
创业:17
中国:16
特别:16
技术:15
```

2.8　本章小结

本章从 Lucene 的概述开始讲起，重点包括 Lucene 分词、索引文档、搜索文档和搜索高亮 4 个部分。本章的每个示例都给出了代码和运行结果，“纸上得来终觉浅，绝知此事要躬行”，希望读者能够动手实践，掌握 Lucene 基础。

第 3 章

Lucene 文件检索项目实战

本章学习要点：

* 需求分析
* 文本内容抽取
* 文件的索引
* 文件搜索
* 文件下载

3.1 需求分析

相信大家一定使用过百度文库、360 文库等互联网产品，以百度文库为例，当我们需要查找一些参考文档时，会习惯性地打开浏览器，输入需求关键词“百度一下”，服务器就会返回一系列相关文档供我们阅读、下载。如图 3-1 所示，返回的文档格式有多种，常见的有 Word 文档（DOC、DOCX）、演示文稿（PPT、PPTX）、文本文件（TXT）、PDF 文档（PDF）、表格（XLS）这几种，搜索结果还会把关键字高亮显示。

假设现在有一批文档，文档格式有 DOC、DOCX、PPT、PPTX、TXT、PDF 这几种，现在要实现一个类似百度文库的文件检索系统。经分析，项目需求如下：

（1）能够对文件名进行检索。

（2）能够对文件内容进行检索。

（3）能够下载检索到的文件。

（4）能够实现关键字的高亮。

第 2 章介绍了 Lucene 开发的入门知识，本章通过文件检索这个小项目整合前面的基础知识。



图 3-1　百度文库搜索结果页面

3.2　架构设计

文件检索系统的架构设计如图 3-2 所示，简单概括如下：文件存储系统中存放了不同类型的文件，后台通过程序提取出文件名和文档内容，使用 Lucene 对文件名和文档内容进行索引，前端对用户提供查询接口，用户提交关键词之后检索索引库，返回匹配文档至前端页面。

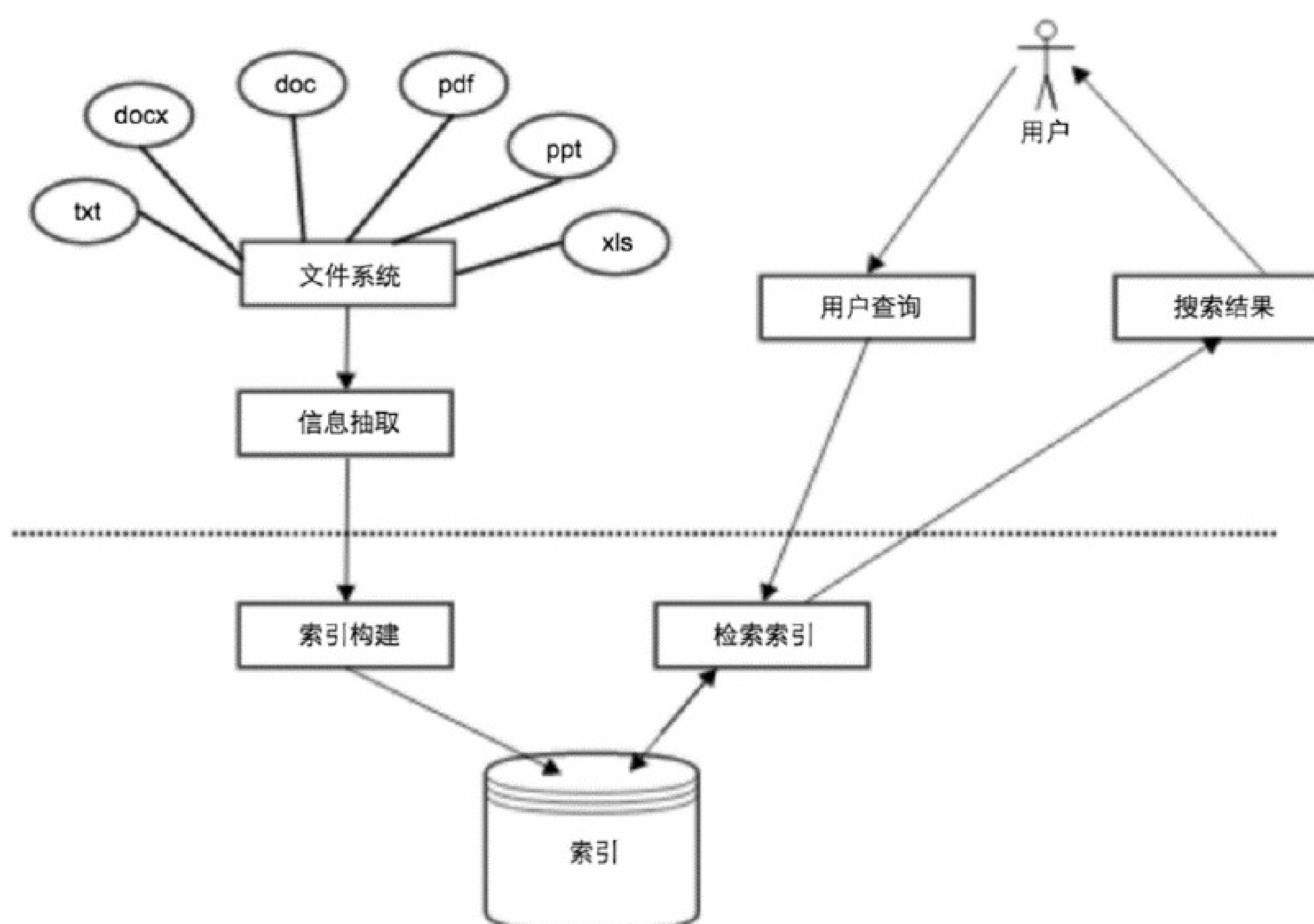


图 3-2　文件检索系统的架构设计图

按照图 3-2 所示的系统架构图，准备一些测试文档作为要检索的文件系统，使用开源工具 Tika 完成信息抽取，使用 Lucene 构建索引，使用 JSP 页面给用户提供查询接口，使用 Servlet 完成搜索，构建类百度文库的小型文件检索系统。

3.3　文本内容抽取

做文件搜索项目面临的第一个难题是如何从各种各样的文件中抽取出来内容，只有抽取出来内容才能进行索引和搜索。本节先来介绍内容解析提取工具 Apache Tika。

3.3.1　Tika 简介

Apache Tika 是一个用于文件类型检测和文件内容提取的库，该项目于 2007 年 3 月开始启动，最开始是 Apache Lucene 项目的子项目，2010 年 5 月成为 Apache 组织的顶级项目，该项目的目标使用群体主要为搜索引擎以及其他内容索引和分析工具，编程语言为 Java。Tika 可以检测超过 1000 种不同类型的文档，比如 PPT、PDF、DOC、XLS 等，所有的文本类型都可以通过一个简单的接口被解析，Tika 广泛应用于搜索引擎、内容分析、文本翻译、数字资产管理等诸多领域。

为什么要使用 Tika？据 filext.com 网站统计，文件内容类型有几万种之多，并且这个数字还在与日俱增。数据有不同的格式，如文本文档、Excel 表格、PDF、图像和多媒体文件等，应用程序如搜索引擎和内容管理系统需要从这些不同类型文档中容易地提取数据。Tika 通过提供一个通用的 API 来检测并提取多种文件格式的内容服务来达到这一目的。Tika 的特点如下：

- 统一解析器接口。Tika 所有的第三方解析器库被封装在一个单一的解析器中，由于这个特征，用户减少了根据不同文档类型选择合适的解析器库的负担。
- 低内存占用。因为统一的解析器接口，Tika 消耗的内存资源更少，也很容易嵌入各种 Java 应用程序中。
- 快速处理。应用中内容检测和信息提取可以预期，处理速度快。
- 灵活元数据。Tika 理解所有用来描述文件的元数据模型。
- 解析器集成。Tika 可以使用单一应用程序中每个文件类型的各种解析器库。
- MIME 类型检测。Tika 可以检测并从所有包括在 MIME 标准的媒体类型中提取内容。
- 语言检测。Tika 包括语言识别功能，因此可以在一个多语种网站基于语言类型的文档中使用。

3.3.2　Tika 下载

下面通过实例来看如何使用 Tika 这一利器。首先到 Tika 官网的下载页面（https://tika.apache.org/download.html）下载相应的 jar 包，可以看到当前版本为 1.13，单击“Mirrors for tika-app-1.13.jar”之后选择合适的镜像源下载 Tika 的 jar 包。

Tika 可作为 GUI 工具使用。打开终端（Windows 平台打开 CMD 命令行工具），切换到 tika-app-1.13.jar 所在目录，启动 Tika，运行命令：

```
java -jar tika-app-1.13.jar -g
```

启动成功以后，Tika 客户端界面如图 3-3 所示。

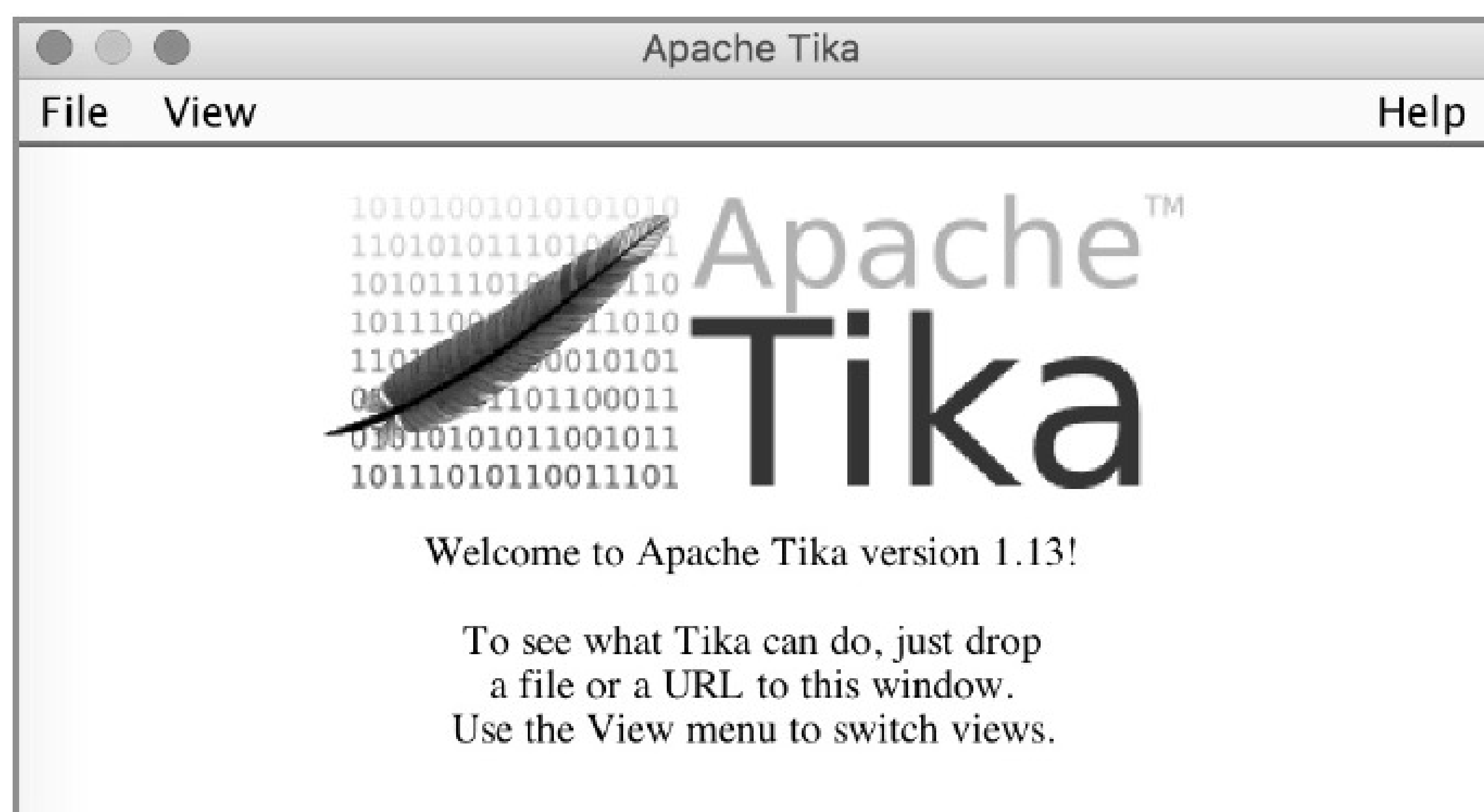


图 3-3　Tika 启动界面

单击 File 菜单按钮，可以选择打开一个本地文件，也可以打开一个远程 URL 地址。文件打开以后，默认情况下显示的是文本的元数据信息，如图 3-4 所示。如果想查看更多内容，可以单击 View 菜单，分别查看文档的元数据信息、格式化之后的文本内容、纯文本内容、核心内容、结构化文本内容以及递归转换后的 JSON 文本。

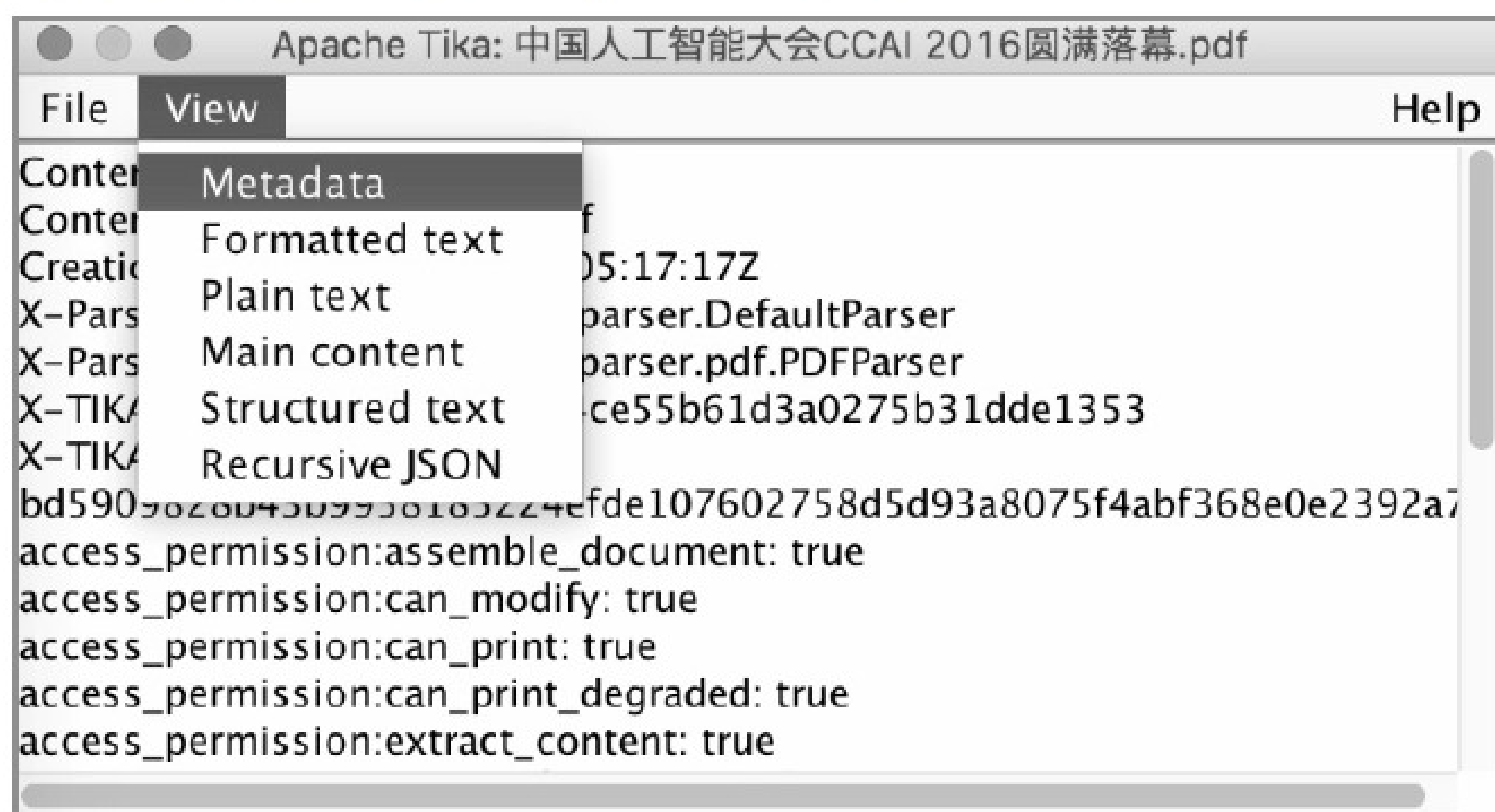


图 3-4　Tika 解析 PDF 文件

3.3.3　搭建工程

这一小节介绍如何在 Java 程序中使用 Tika，创建 Java 工程的步骤如下：

步骤 01 打开 Eclipse，新建 java project，把工程命名为 TikaDemo。

步骤 02 在工程根目录下新建一个 lib 文件夹，拷贝 tika-app-1.13.jar 到 lib 目录下。

步骤 03 选中 tika-app-1.13.jar 并右击，在打开的快捷菜单中依次选择 Build Path→Add to Build Path，把 jar 包加入工程类路径下，然后就可以在 java 类中调用 Tika 提供的各种 API。

步骤 04 添加完 jar 包以后在工程根目录下新建一个 files 文件夹用于存放测试文件，这里我们在 files 文件夹下放了 DOC、DOCX、PDF、TXT、PPTX 5 种类型的文件。

步骤 05 最后，新建一个名为 TikaParsePdf 的 Java 类。

上述步骤完成以后，工程目录结构如图 3-5 所示。

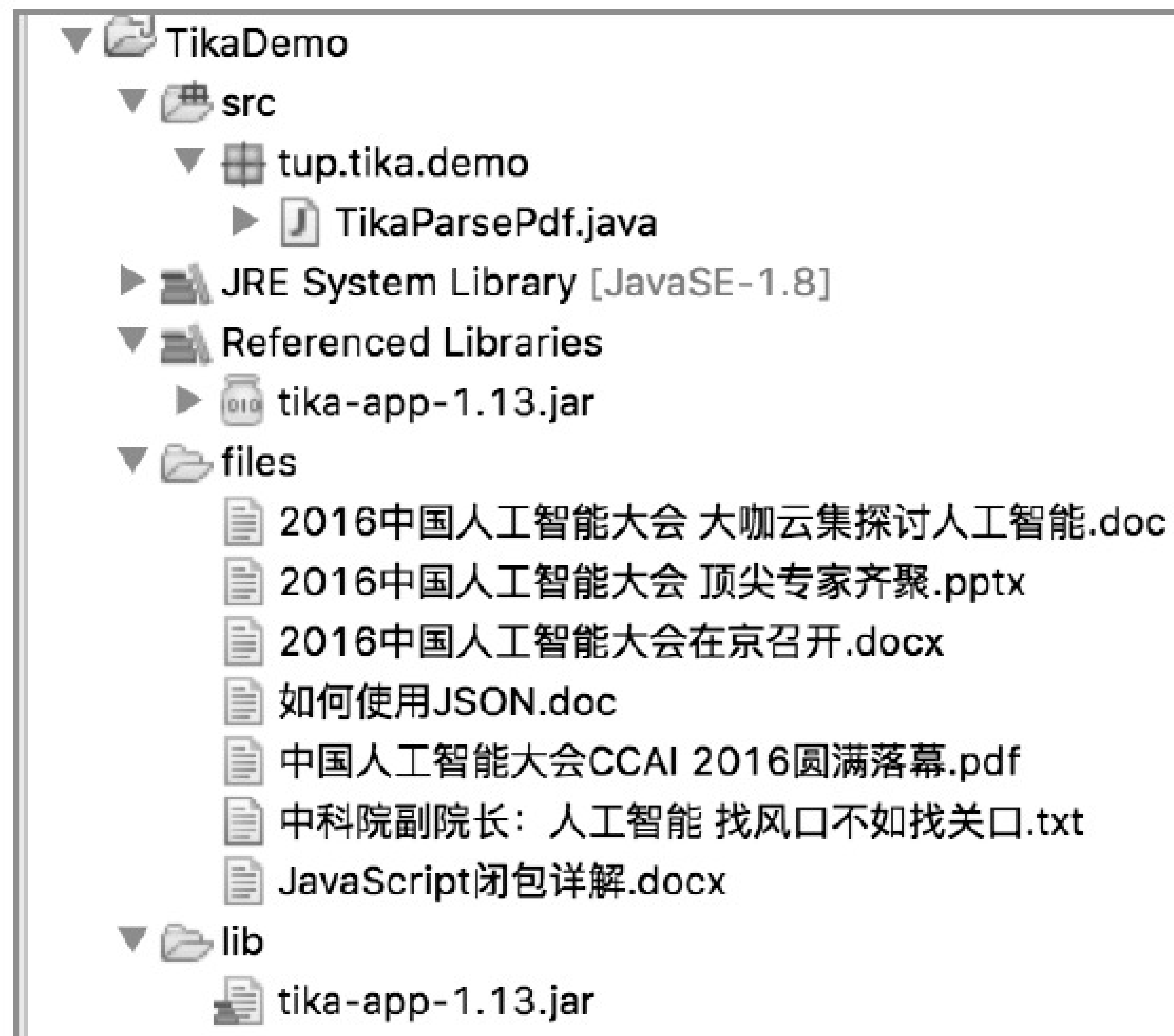


图 3-5　Tika 工程目录结构

3.3.4　内容抽取

在 TikaParsePdf.java 中编写 Java 代码，提取出 PDF 文件中的文本内容，最后打印到控制台，见代码清单 3-1。

代码清单 3-1　Tika 提取 PDF 文件内容

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import org.apache.tika.exception.TikaException;
import org.apache.tika.metadata.Metadata;
import org.apache.tika.parser.ParseContext;
import org.apache.tika.parser.pdf.PDFParser;
import org.apache.tika.sax.BodyContentHandler;
import org.xml.sax.SAXException;

public class TikaParsePdf {
  public static void main(String[] args) throws IOException,
    SAXException, TikaException {
    // 文件路径
    String filepath = "files/中国人工智能大会 CCAI 2016 圆满幕.pdf";
    // 新建 File 对象
    File pdfFile = new File(filepath);
    // 创建内容处理器对象
    BodyContentHandler handler = new BodyContentHandler();
```

```
            // 创建元数据对象
            Metadata metadata = new Metadata();
            // 读入文件
            FileInputStream inputStream = new FileInputStream(pdfFile);
            // InputStream inputStream=TikaInputStream.get(pdfFile);
            // 创建内容解析器对象
            ParseContext parseContext = new ParseContext();
            // 实例化 PDFParser 对象
            PDFParser parser = new PDFParser();
            // 调用 parse()方法解析文件
            parser.parse(inputStream, handler, metadata, parseContext);
            // 遍历元数据内容
            System.out.println("文件属性信息:");
            for (String name : metadata.names()) {
                System.out.println(name + ":" + metadata.get(name));
            }
            // 打印 pdf 文件中的内容
            System.out.println("pdf 文件中的内容:");
            System.out.println(handler.toString());
        }
    }
```

运行结果如图 3-6 所示。

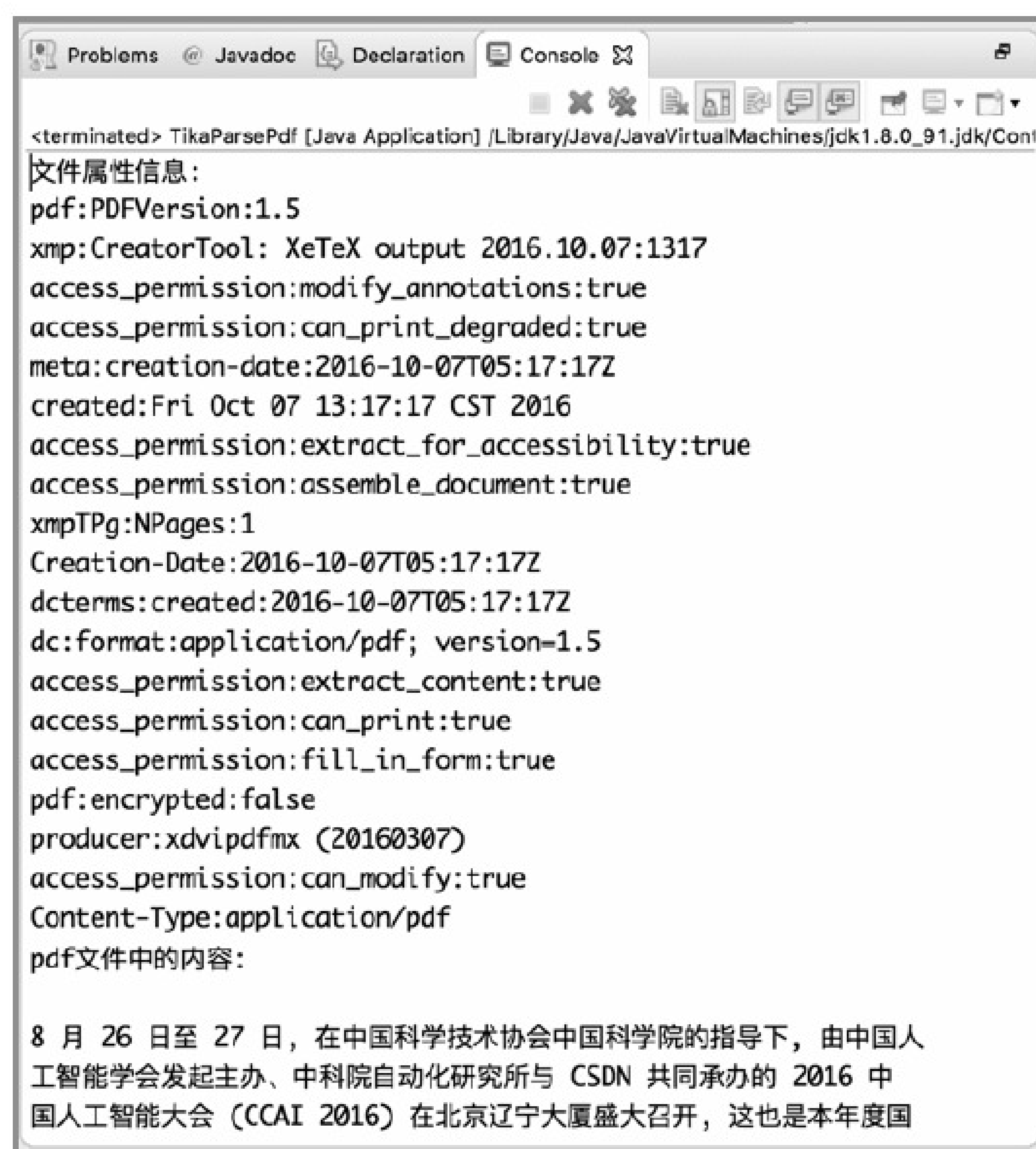


图 3-6　Tika PDF 文件运行结果

上面的例子实现了如何使用 Tika 读取本地文件中的内容，files 目录下放了一个 PDF 文件用于做测试，在主函数中首先创建一个 File 对象 pdfFile，传入参数为 PDF 文件的路径，通过

文件路径作为参数实例化一个文件对象，这样一来在程序中 pdfFile 就指向了 PDF 文件。目标是要提取的文件的文本内容，因此需要实例化 BodyContentHandler 对象，创建 Metadata 对象用于获取文件属性。之后实例化一个 FileInputStream 对象，将 pdfFile 对象作为参数传给 FileInputStream 类的构造方法，但是使用这种输入流不支持随机访问读取文件，如果想要更高效地处理各种类型的文件，可以使用 Tika 提供的 TikaInputStream 类。接下来创建一个解析上下文的 ParseContext 对象并实例化一个 PDF 解析器对象，然后调用 PDF 解析器对象的 parse 方法，并传入所有需要的 4 个参数，包含任何文件内容的 InputStream 对象、ContentHandler 对象、metadata 元数据对象和 ParseContext 对象。解析完成以后可以通过内容处理器对象（本实例中的 handler）的 toString()方法输出文件内容，文件属性名保存在元数据对象的 names()方法中，返回结果是字符数组，遍历属性名数组通过元数据对象的 get()方法得到属性信息。

上面的例子是如何提取 PDF 格式的文件内容以及元数据信息，那么该如何处理其他类型的文件呢？我们注意到，在创建解析器对象的时候使用的是 PDFParser 类，PDFParser 类适用于解析 PDF 格式的文件，如果想解析其他类型的文件，就需要更改解析器接口类型。下面一一列出创建解析不同类型文件所需要的解析器对象的方法：

- 解析 MS Office 文档：

```
OOXMLParser  parser = new OOXMLParser ();
```

- 解析文本文件：

```
TXTParser  parser = new TXTParser();
```

- 解析 HTML 文件：

```
HtmlParser parser = new HtmlParser();
```

- 解析 XML 文件：

```
XMLParser parser = new XMLParser();
```

- 解析 class 文件：

```
ClassParser  parser = new  ClassParser();
```

读者可以把 TikaParsePdf.java 中的文件路径加以修改，根据文档类型采用不同的解析器接口进行实验。除了解析常见的文档类文件，Tika 还可以解析图像、音频（如 MP3）、视频（如 MP4）等多种类型的文件。

3.3.5　自动解析

上述解析 PDF 的例子的流程大致如下：确定要解析 PDF 文件→实例化 PDFParser→提取内容，如果要提取的文档类型不止一种，又该如何处理？Tika 的强大之处正在于此，它可以先判断文档类型，再根据文档类型实例化解析器接口，流程如图 3-7 所示。

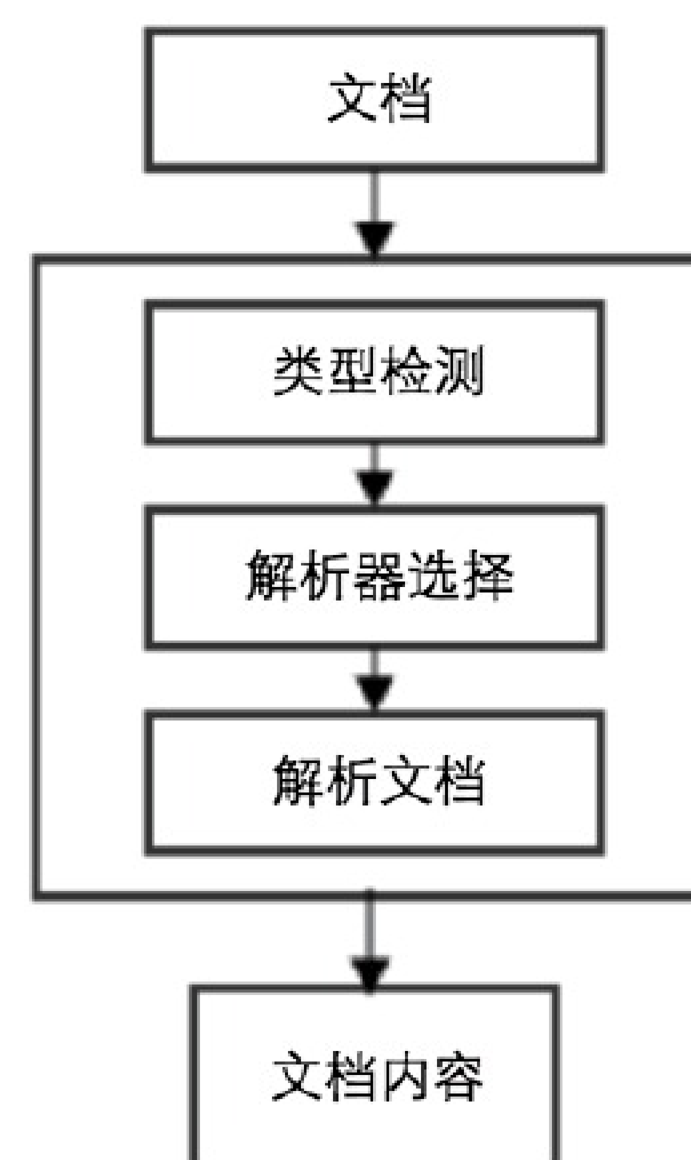


图 3-7　文档自动解析流程

自动解析文档的分析过程如下：

（1）首先，传入一个文件到 Tika，文件类型可以是任意的，Tika 使用自身的类型检测机制来检测文件类型。

（2）Tika 提供了一个解析器库，解析器库中包含多种类型的解析器。一旦文档类型被检测出来，下一步就可以根据已知的文档类型从解析器库中选择合适的解析器。

（3）选择合适的解析器以后就可以把文档传送到解析器中，解析完成后即可进行文档内容提取、元数据提取。

下面介绍使用 Tika 对象和使用 Parser 接口两种方法进行内容提取。使用 Tika 对象提取文档内容的 Java 代码见代码清单 3-2。

代码清单 3-2　使用 Tika 对象提取文档内容

```
import java.io.File;
import java.io.IOException;
import org.apache.tika.Tika;
import org.apache.tika.exception.TikaException;
public class TikaExtraction {
    public static void main(String[] args)
       throws IOException, TikaException {
        Tika tika = new Tika();
        // 新建存放各种文件的 files 文件夹
        File fileDir = new File("files");
        // 如果文件夹路径错误，退出程序
        if (!fileDir.exists()) {
            System.out.println("文件夹不存在，请检查!");
            System.exit(0);
        }
        // 获取文件夹下的所有文件，存放在 File 数组中
        File[] fileArr = fileDir.listFiles();
        String filecontent;
        for (File f : fileArr) {
            filecontent = tika.parseToString(f);// 自动解析
        System.out.println("Extracted Content: " + filecontent);
      }
    }
}
```

上述代码首先新建了一个 File 对象指向存放各种文档的文件夹，通过 File 对象的 exists()方法判断文件夹路径是否正确，如果路径错误则退出程序，打印提示信息。接下来，通过 listFiles()方法获取 files 目录下所有的文件，存放在文件数组之中。最后新建一个 Tika 对象，调用 parseToString()方法获取文档内容，该方法的传入参数为 File 对象。

使用 Tika 对象提取文档内容的 Java 代码见代码清单 3-3。

代码清单 3-3　使用 Parser 接口提取文档内容

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import org.apache.tika.exception.TikaException;
import org.apache.tika.metadata.Metadata;
import org.apache.tika.parser.AutoDetectParser;
import org.apache.tika.parser.ParseContext;
import org.apache.tika.parser.Parser;
import org.apache.tika.sax.BodyContentHandler;
import org.xml.sax.SAXException;
public class ParserExtraction {
public static void main(String[] args) throws IOException,
    SAXException, TikaException {
    // 新建存放各种文件的 files 文件夹
    File fileDir = new File("files");
    // 如果文件夹路径错误，退出程序
    if (!fileDir.exists()) {
        System.out.println("文件夹不存在，请检查!");
        System.exit(0);
    }
    // 获取文件夹下的所有文件，存放在 File 数组中
    File[] fileArr = fileDir.listFiles();
    // 创建内容处理器对象
     BodyContentHandler handler = new BodyContentHandler();
    // 创建元数据对象
     Metadata metadata = new Metadata();
     FileInputStream inputStream = null;
     Parser parser = new AutoDetectParser();
     // 自动检测分词器
     ParseContext context = new ParseContext();
     for (File f : fileArr) {
        inputStream = new FileInputStream(f);
        parser.parse(inputStream, handler, metadata, context);
        System.out.println(f.getName() + ":\n" + handler
                          .toString());
    }
}
}
```

使用 Parse 接口自动提取内容和前面单一的提取一种文档的区别在于实例化对象不一样，AutoDetectParser 是 CompositeParser 的子类，它能够自动检测文件类型，并使用相应的方法把接收到的文档自动发送给最接近的解析器类。

3.4　工程搭建

通过 3.3 节的学习，文件内容抽取问题已经解决了。下面介绍如何从零开始构建文件检索系统，在开始之前确保计算机已经正确安装 Java、Eclipse、Apache Tomcat。

步骤01 在 Eclipse 中新建一个 Java Web 工程。启动 Eclipse，单击 File->New->Dynamic Web Project，工程名命名为 filesearch，如图 3-8 所示。

步骤02 单击 Target runtime 下方的"New Runtime"按钮设置 Tomcat 路径，如图 3-9 所示。我们选择 Apache Tomcat 7.0，找到 Tomcat 所在位置，选定 JRE 版本为 1.8，最后单击 Finish。

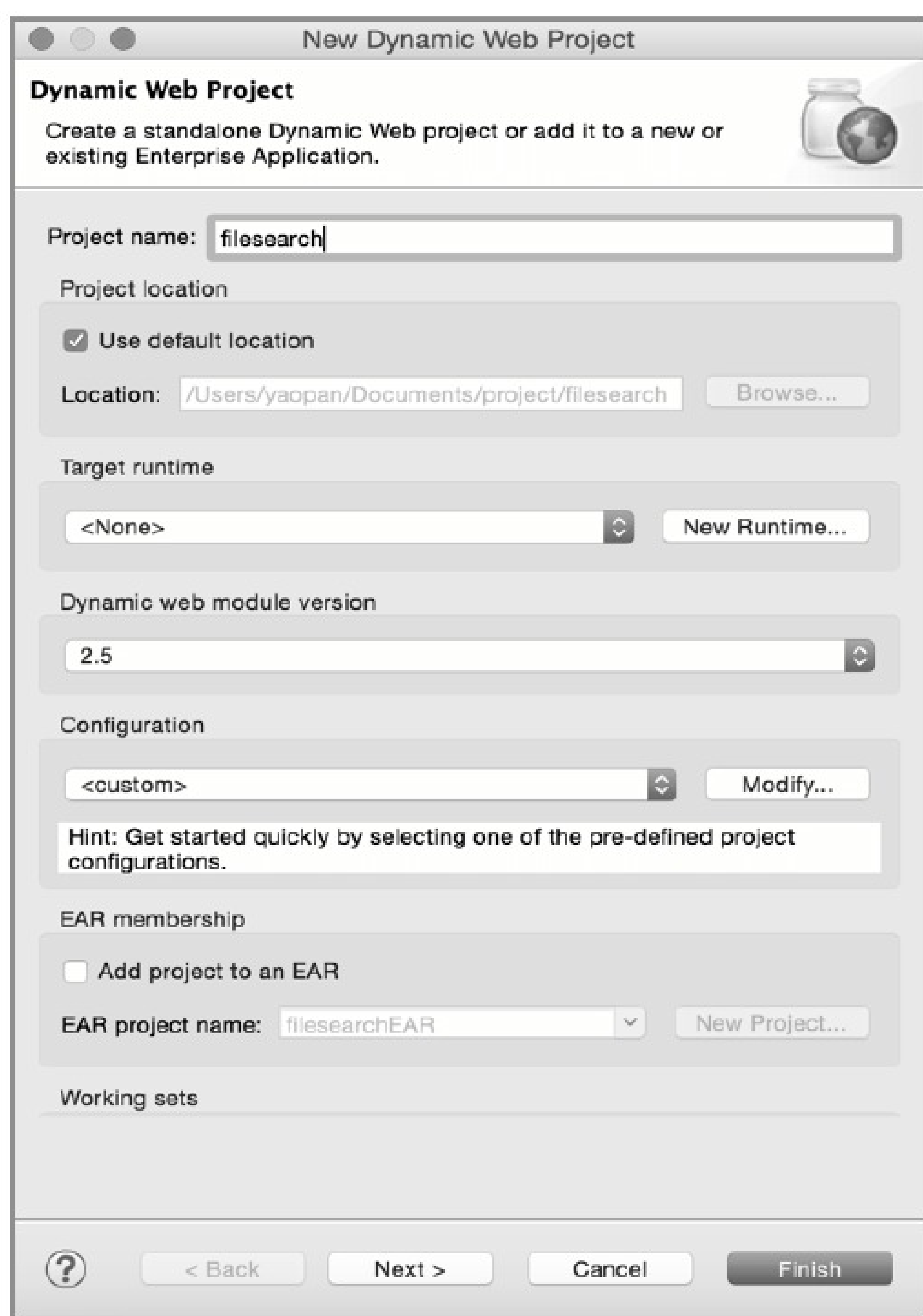


图 3-8　新建一个 Java Web 工程

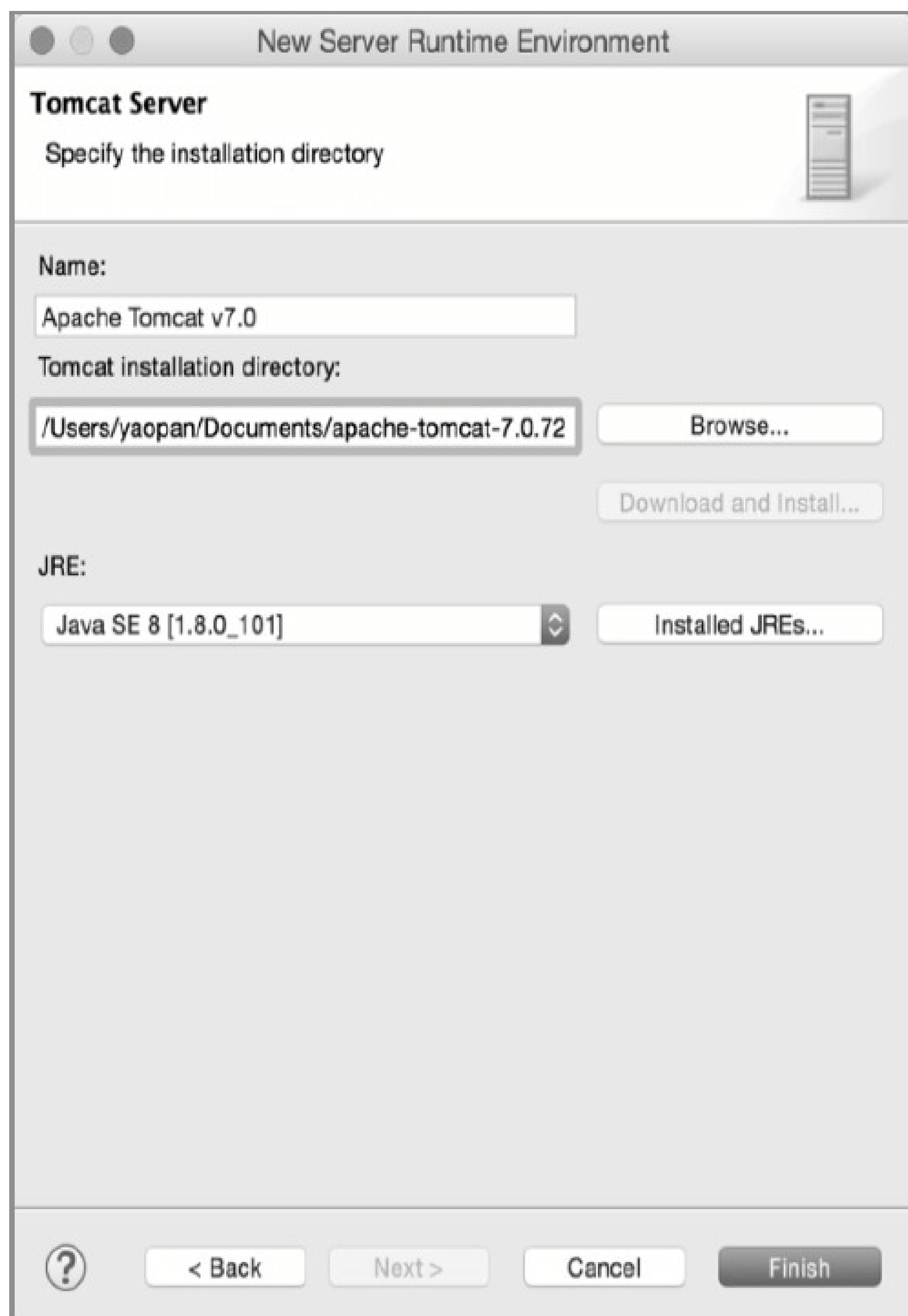


图 3-9　设置 Tomcat 路径

步骤03 设置完运行环境以后，下面的 Dynamic web module version 选择 2.5，选择 2.5 版本的会在 WebContent 的 WEB-INF 目录下自动生成 web.xml。

步骤04 单击 Finish 按钮，一个 Java Web 项目创建完成。

步骤05 在 Tomcat 中运行 Web 工程。

WebContent 目录下新建一个 index.jsp，在 body 标签之间加上一个 h1 标签并输入"Hello Lucene"字符串作为首页的提示信息。然后选中工程名并右击，依次选择 Run As→Run on Server→Tomcat v7.0 Server at localhost→Finish，待服务器启动以后打开浏览器访问：http://localhost:8080/filesearch，运行效果如图 3-10 所示。

步骤06 添加 jar 包。拷贝下面这些 jar 包到 filesearch/WebContent/WEB-INF/lib：IKAnalyzer2012_u6.jar、lucene-analyzers-common-6.0.0.jar、lucene-analyzers-smartcn-6.0.0.jar、lucene-core-6.0.0.jar、lucene-highlighter-6.0.0.jar、lucene-memory-6.0.0.jar、lucene-queries-6.0.0.jar、lucene-queryparser-6.0.0.jar、tika-app-1.13.jar

步骤07 新建包和资源文件夹。在 Web 工程的 src 目录下新建 3 个包，分别命名为 lucene.file.search.controller、lucene.file.search.model、lucene.file.search.service。lucene.file.search.controller 包中主要存放 Servlet 控制器，lucene.file.search.model 用于存放实体类，lucene.file.search.service 用于存放工具类。然后，在 WebContent 目录下新建一个名为 css 的文件夹用于存放样式表文件，新建一个名为 files 的文件夹用于存放要检索的各种类型的文档，新建一个名为 images 的文件夹用于存放图片资源，新建一个名为 indexdir 的文件夹用于存放索引库。最后在 files 目录下放置一些文档用于测试，即被搜索的对象。本实例中的文档格式有 DOC、DOCX、PPTX、PDF、TXT 5 种。

经过以上几个步骤，Lucene 文件检索系统的环境已经搭建完成，工程目录如图 3-11 所示。



图 3-10　Web 工程首页信息

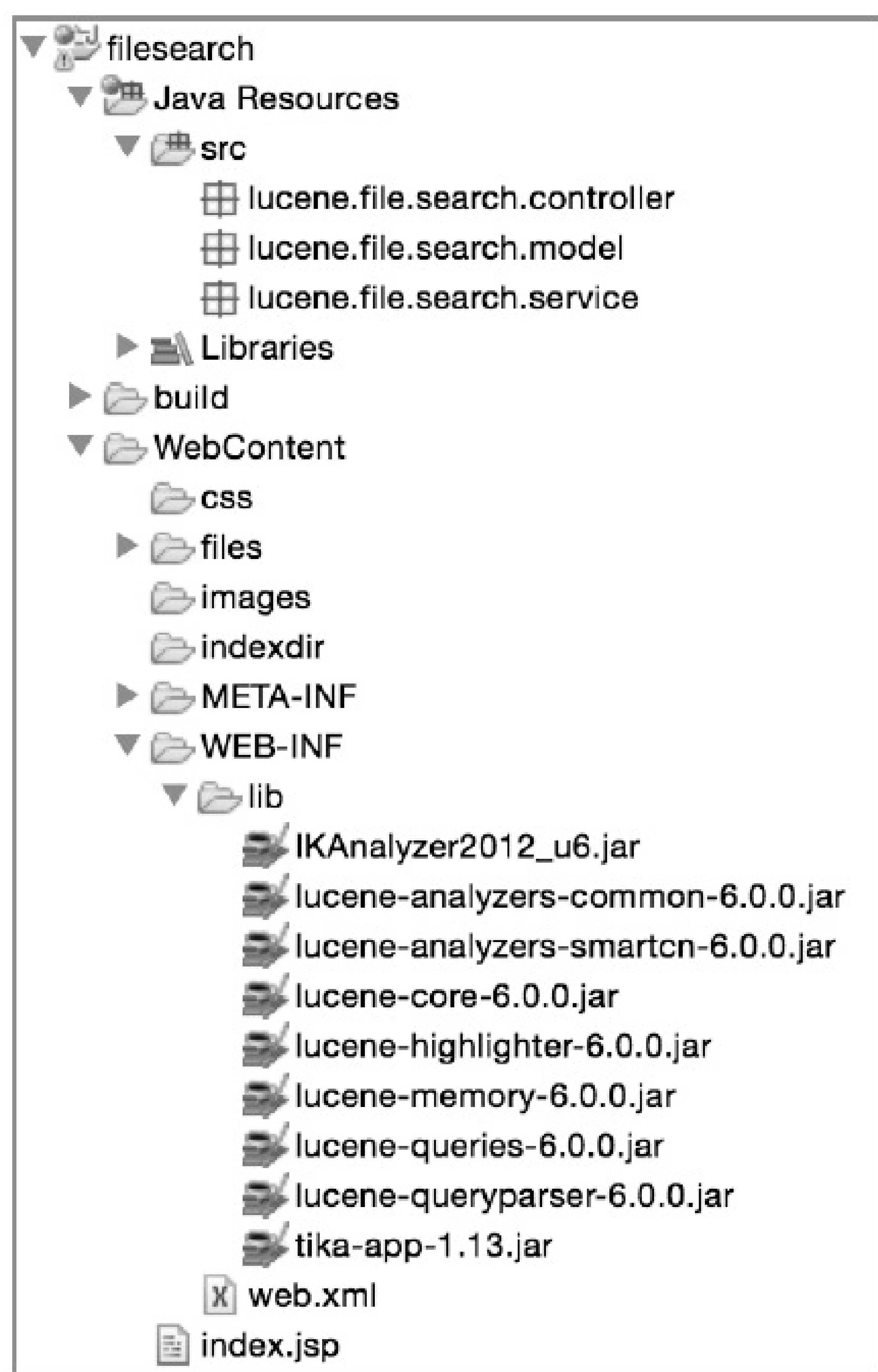


图 3-11　工程目录

3.5　索引文档

工程搭建完成以后，首先进行索引的构建。要检索的对象是文件，为了简单起见，我们只索引文档名和文档内容。利用 Java 面向对象的思想，在 lucene.file.search.model 目录下新建一个实体类，类名为 FileModel，表示文件对象，设置 title 和 content 两个 String 类型的成员变量并提供对应的 setter 和 getter 方法，最后添加一个有参构造方法，示例见代码清单 3-3。

代码清单 3-3　使用 Parser 接口提取文档内容

```
package lucene.file.search.model;
public class FileModel {
        private String title;// 文件标题
        private String content;// 文件内容
        public String getTitle() {
            return title;
        }
        public void setTitle(String title) {
            this.title = title;
        }
        public String getContent() {
            return content;
        }
        public void setContent(String content) {
            this.content = content;
        }
        public FileModel() {
        }
        public FileModel(String title, String content) {
            this.title = title;
            this.content = content;
        }
}
```

在 lucene.file.search.service 目录下新建一个创建索引的 Java 类，类名为 CreateIndex，在类中添加两个静态方法：extractFile()方法和 ParserExtraction()方法。extractFile()方法用于列出 WebContent/files 目录下的所有文件，返回值类型为 FileModel 类型的列表。Java 代码如下：

```
public static List<FileModel> extractFile() throws IOException {
    ArrayList<FileModel> list = new ArrayList<FileModel>();
    File fileDir = new File("WebContent/files");
    File[] allFiles = fileDir.listFiles();
    for (File f : allFiles) {
    FileModel sf = new FileModel(f.getName(), ParserExtraction(f));
        list.add(sf);
    }
    return list;
}
```

ParserExtraction()方法的功能是使用 Tika 提取文档内容，传入参数为一个 File 对象。Java 代码如下：

```
public static String ParserExtraction(File file) {
        String fileContent = "";//接收文档内容
        BodyContentHandler handler = new BodyContentHandler();
        Parser parser = new AutoDetectParser();//自动解析器接口
        Metadata metadata = new Metadata();
        FileInputStream inputStream;
        try {
            inputStream = new FileInputStream(file);
            ParseContext context = new ParseContext();
            parser.parse(inputStream, handler, metadata, context);
            fileContent = handler.toString();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (SAXException e) {
            e.printStackTrace();
        } catch (TikaException e) {
            e.printStackTrace();
        }
        return fileContent;
}
```

分词器使用 IK 分词，把 IKTokenizer6x.java 和 IKAnalyzer6x.java 放到 lucene.file.search.service 包下（Lucene 6.0 中使用 IK 分词器的方法参考第 2 章）。CreateIndex 类中新建主函数，代码如下：

```
public static void main(String[] args) throws IOException {
        // IK 分词器对象
        Analyzer analyzer = new IKAnalyzer6x();
        IndexWriterConfig icw = new IndexWriterConfig(analyzer);
        icw.setOpenMode(OpenMode.CREATE);
        Directory dir = null;
        IndexWriter inWriter = null;
        Path indexPath = Paths.get("WebContent/indexdir");
        FieldType fileType = new FieldType();
        fileType.setIndexOptions(IndexOptions.
            DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS);
        fileType.setStored(true);
        fileType.setTokenized(true);
        fileType.setStoreTermVectors(true);
        fileType.setStoreTermVectorPositions(true);
        fileType.setStoreTermVectorOffsets(true);
```

```
        Date start = new Date();// 开始时间
        if (!Files.isReadable(indexPath)) {
            System.out.println(indexPath.toAbsolutePath() + "不存在或
                者不可读，请检查!");
            System.exit(1);
        }
        dir = FSDirectory.open(indexPath);
        inWriter = new IndexWriter(dir, icw);
        ArrayList<FileModel> fileList = (ArrayList<FileModel>)
        extractFile();
        // 遍历 fileList,建立索引
        for (FileModel f : fileList) {
            Document doc = new Document();
            doc.add(new Field("title", f.getTitle(), fileType));
            doc.add(new Field("content", f.getContent(), fileType));
            inWriter.addDocument(doc);
        }
        inWriter.commit();
        inWriter.close();
        dir.close();
        Date end = new Date();// 结束时间
        // 打印索引耗时
        System.out.println("索引文档完成,共耗时:" + (end.getTime() -
            start.getTime())+ "毫秒.");
    }
```

运行 CreateIndex 类中的 main 方法，/filesearch/WebContent/files 目录下的所有文档，不论是什么格式，文件名和文件内容都被写入到了 Lucene 索引。

3.6　查询界面

索引构建完成以后，我们来看看如何实现前端的用户接口，换言之就是在后台接收用户的查询关键词。编辑 index.jsp，在 index.jsp 中加入一个 form，包含一个输入框和一个提交按钮。form 中可以设置参数，action="SearchFile"表示单击提交按钮之后输入框中的内容会被名为 SearchFile 的 Servlet 接收，method="get"表示使用的是 HTTP 的 get 方法。SearchFileServlet 我们会在后面实现，先使用前端技术实现一个简单的搜索界面，为了界面的美观，加了一个 logo 并使用 CSS 调整页面样式。head 标签中的<link type="text/css" rel="stylesheet" href="css/index.css">一行用于引入 WebContent/css 目录下的 CSS 文件。index.jsp 的代码见代码清单 3-4。

代码清单 3-4

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" import="java.util.Calendar"%>
<%
    Calendar cal = Calendar.getInstance();
    int year = cal.get(Calendar.YEAR); //获取年份
%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>lucene 文件检索</title>
    <link type="text/css" rel="stylesheet" href="css/index.css">
    </head>
    <body>
    <div class="indexbox">
    <div class="logo">
    <a href="index.jsp"><img alt="文件检索" src="images/logo.png"> </a>
    </div>
      <div class="searchform">
        <form action="SearchFile" method="get">
          <input type="text" name="query">
          <input type="submit" value="搜索">
        </form>
        </div>
        <div class="info">
            <p>基于 Lucene 的文件检索系统</p>
            <br />
            <p>&copy; <%=year > 2016 ? (2016 + "-" + year) : year%>
              清华大学出版社 All rights Reserved</p>
            </div>
            </div>
        </body>
    </html>
```

CSS 样式的内容不在本书的讲解范围之内，读者可以到本项目的源代码中查看 CSS 的内容，代码不在此给出。重启 Tomcat 服务器，首页界面效果如图 3-12 所示。



图 3-12　用户查询界面

3.7　文件检索

用户在查询界面提交的查询关键词需要被后台接收，接收任务是由 Servlet 完成的。在 lucene.file.search.controller 包下新建一个 Java 类，类名为 SearchFileServlet，继承 HttpServlet 类并重写 doGet 和 doPost 方法。代码如下：

```
public class SearchFileServlet extends HttpServlet {
protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        System.out.println("SearchFileServlet");
    }
protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        doGet(request, response);
    }
}
```

修改 web.xml，在<web-app> </web-app>标签中添加一下配置：

```
    <servlet>
        <description>SearchFileServlet</description>
        <display-name>SearchFileServlet</display-name>
        <servlet-name>SearchFileServlet</servlet-name>
        <servlet-class>
            lucene.file.search.controller.SearchFileServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SearchFileServlet</servlet-name>
```

```
        <url-pattern>/SearchFile</url-pattern>
    </servlet-mapping>
```

<servlet>元素是用于配置信息，指定 servlet-name 为 SearchFileServlet，对应的类路径为 lucene.file.search.controller.SearchFileServlet。<servlet-mapping>元素是在 Servlet 和 URL 样式之间定义一个映射，即 servlet 类提供一个 url，在地址栏输入对应的映射路径就可以访问对应的 servlet。我们在 doGet 方法中输出字符提示信息用于测试 Servlet 是否配置成功，之后重启服务器，访问 http://localhost：8080/filesearch/SearchFile，Eclipse 的控制台中输出“SearchFileServlet”提示，说明 Servlet 配置成功。

Servlet 配置正确以后，用户在查询界面提交查询关键词以后，就可以在 SearchFileServlet 类中接收参数进行搜索。在 SearchFileServlet.java 中添加一个搜索文档的方法，返回结果为 FileModel 类型的 list，传入参数为查询关键词、索引路径、返回文档的条数。代码如下：

```
public static ArrayList<FileModel> getTopDoc(String key, String
    indexpathStr,int N)      {
    ArrayList<FileModel> hitsList = new ArrayList<FileModel>();
    //检索域
    String[] fields = { "title", "content" };
    Path indexPath = Paths.get(indexpathStr);
    Directory dir;
    try {
      dir = FSDirectory.open(indexPath);
      IndexReader reader = DirectoryReader.open(dir);
      IndexSearcher searcher = new IndexSearcher(reader);
      Analyzer analyzer = new IKAnalyzer6x();
      MultiFieldQueryParser parser2 = new
        MultiFieldQueryParser(fields, analyzer);
      // 查询字符串
      Query query = parser2.parse(key);
      TopDocs topDocs = searcher.search(query, N);
      // 定制高亮标签
      SimpleHTMLFormatter fors = new SimpleHTMLFormatter("<span
        style=\"colo r:red;\">", "</span>");
      QueryScorer scoreTitle = new QueryScorer(query, fields[0]);
      Highlighter hlqTitle = new Highlighter(fors, scoreTitle);
      QueryScorer scoreContent = new QueryScorer(query, fields[0]);
      Highlighter hlqContent = new Highlighter(fors, scoreTitle);
      TopDocs hits = searcher.search(query, 100);
      for (ScoreDoc sd : topDocs.scoreDocs) {
            Document doc = searcher.doc(sd.doc);
            String title = doc.get("title");
            String content = doc.get("content");
            TokenStream tokenStream = TokenSources.getAnyTokenStream
```

```
            (searcher.getIndexReader(), sd.doc, fields[0],
                        new IKAnalyzer6x());// 获取 tokenstream
        Fragmenter fragment = new SimpleSpanFragmenter(scoreTitle);
        hlqTitle.setTextFragmenter(fragment);
        String hl_title = hlqTitle.getBestFragment(tokenStream, title);
        // 获取高亮的片段，可以对其数量进行限制
        tokenStream = TokenSources.getAnyTokenStream(searcher.
         getIndexReader(), sd.doc, fields[1],new IKAnalyzer6x());
        fragment = new SimpleSpanFragmenter(scoreContent);
        hlqContent.setTextFragmenter(fragment);
        String hl_content = hlqTitle.getBestFragment(tokenStream,
            content);// 获取高亮的片段，可以对其数量进行限制
        FileModel fm = new FileModel(hl_title != null ? hl_title :
                    title,hl_content != null ? hl_content : content);
                hitsList.add(fm);
        }
            dir.close();
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        } catch (InvalidTokenOffsetsException e) {
            e.printStackTrace();
        }
        return hitsList;
    }
```

继续完善 doGet 方法，先使用 request 对象的 getParameter()方法接收 index.jsp 中传来的表单字符串，如果查询字符串接收正确则调用 getTopDoc()方法，把查询结果放到 request 作用域中；如果查询字符串为空则转到错误页面。完整的 doGet 方法如下：

```
protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    //索引路径
    String indexpathStr = request.getServletContext()
        .getRealPath ("/indexdir");
    //接收查询字符串
    String query = request.getParameter("query");
    //编码格式转换
    query = new String(query.getBytes("iso8859-1"), "UTF-8");
    if (query.equals("") || query == null) {
        System.out.println("参数错误!");
        request.getRequestDispatcher("error.jsp")
```

```
            .forward(request, response);
        } else {
        ArrayList<FileModel> hitsList = getTopDoc(query, indexpathStr,100);
        System.out.println("共搜到:" + hitsList.size() + "条数据!");
        request.setAttribute("hitsList", hitsList);
        request.setAttribute("queryback", query);
        request.getRequestDispatcher("result.jsp").forward(request,
            response);
        }
    }
```

3.8　结果展示

在 WebContent 目录下新建 error.jsp，添加提示信息，在<head></head>标签中设置 5 秒后自动跳转到 index.jsp，代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<meta http-equiv="refresh" content="5;url=index.jsp">
<title>Error Page</title>
</head>
<body>
    <h3>没有搜到数据!5 秒后跳转至搜索页!</h3>
</body>
</html>
```

在编辑 result.jsp 页面之前，首先在 lucene.file.search.service 目录下新建一个用于去除 HTML 标签的正则表达式类 RegexHtml.java，delHtmlTag()方法用于去除字符串中的 HTML 标签，代码如下：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexHtml {
    public String delHtmlTag(String line) {
        String regEx_html = "<[^>]+>";
        // 创建 Pattern 对象
        Pattern r = Pattern.compile(regEx_html);
        // 创建 matcher 对象
        Matcher m = r.matcher(line);
```

```
            line = m.replaceAll("");
            return line;
        }
    }
```

用户搜索到文档以后，添加一个文件下载功能，这样用户就可以把文件下载到本地了。在 lucene.file.search.controller 包下新建一个 FileDownloadServlet.java，代码如下：

```
public class FileDownloadServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static final String CONTENT_TYPE = "text/html; charset=GBK";
    public FileDownloadServlet() {
        super();
    }
protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    response.getWriter().append("Served at:")
        .append(request.getContextPath());
    response.reset();
    response.setContentType(CONTENT_TYPE);
    String filename = new String(request.getParameter("filename")
                          .getBytes("iso-8859-1"), "UTF-8");
    System.out.println(filename);
    System.out.println("文件路径:" + request.getServletContext()
                           .getRealPath("/files") + "/" + filename);
    File file = new File(request.getServletContext().
        getRealPath("/files") + "/" + filename);
    System.out.println(file.getPath());
    // 设置 response 的编码方式
    response.setContentType("application/octet-stream");
    // 写明要下载的文件的大小
    response.setContentLength((int) file.length());
    // 解决中文乱码,向客户端发送返回页面的头信息
    // 1.Content-disposition 是 MIME 协议的扩展
    // 2.attachment --- 作为附件下载
    // 3.在客户端将会弹出下载框
    // 4.这个是文件下载的关键代码
    response.setHeader("Content-Disposition","attachment;
    filename="+ new String(filename.getBytes("UTF-8"),"ISO8859-1"));
    // 读出文件到 i/o 流
    FileInputStream fis = new FileInputStream(file);
    BufferedInputStream buff = new BufferedInputStream(fis);
    byte[] b = new byte[1024];// 相当于我们的缓存
    int k = 0;// 该值用于计算当前实际下载了多少字节
```

```
            // 从 response 对象中得到输出流,准备下载
            OutputStream myout = response.getOutputStream();
            // 开始循环下载
            while (-1 != (k = fis.read(b, 0, b.length))) {
                // 将 b 中的数据写到客户端的内存
                myout.write(b, 0, k);
            }
            // 将写入到客户端的内存的数据,刷新到磁盘
            myout.flush();
            fis.close();
            buff.close();
        }
        protected void doPost(HttpServletRequest request,
            HttpServletResponse response)throws ServletException, IOException {
            doGet(request, response);
        }
    }
```

web.xml 新增 FileDownloadServlet 的配置信息：

```
<servlet>
    <description></description>
    <display-name>FileDownloadServlet</display-name>
    <servlet-name>FileDownloadServlet</servlet-name>
    <servlet-class>
            lucene.file.search.controller.FileDownloadServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>FileDownloadServlet</servlet-name>
    <url-pattern>/FileDownloadServlet</url-pattern>
</servlet-mapping>
```

在 result.jsp 页面中，通过 request 的 getAttribute()方法接收 SearchFileServlet 中的查询结果，遍历到页面，见代码清单 3-5。

代码清单 3-5

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"
    import="java.util.ArrayList"
    import="lucene.file.search.model.FileModel"
    import="java.util.regex.*"
    import="lucene.file.search.service.RegexHtml"
    import="java.util.Iterator"%>
```

```
<%
    String path = request.getContextPath();//获取工程根目录
    String basePath = request.getScheme() + "://" + request.
    getServerName()+ ":" + request.getServerPort()+ path + "/";
        String regEx_html = "<[^>]+>";
        // 创建 Pattern 对象
        Pattern r = Pattern.compile(regEx_html);
        // 现在创建 matcher 对象
        RegexHtml regexHtml = new RegexHtml();
        ArrayList<FileModel> hitsList = (ArrayList<FileModel>)
           request.getAttribute("hitsList");
        String queryback = (String) request.getAttribute("queryback");
%>
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="pragma" content="no-cache">
    <meta http-equiv="cache-control" content="no-cache">
    <meta http-equiv="expires" content="0">
    <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
    <meta http-equiv="description" content="This is my page">
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <base href="<%=basePath%>">
    <title>搜索结果</title>
    <link type="text/css" rel="stylesheet" href="css/result.css">
</head>
<body>
    <div class="searchbox">
        <div class="logo">
        <a href="index.jsp"><img alt="文件检索" src="images/ logo.png"></a>
        </div>
        <div class="searchform">
        <form action="SearchFile" method="get">
        <input type="text" name="query" value= "<%=queryback%>">
        <input  type="submit" value="搜索">
        </form>
    </div>
    </div>
    <div class="result">
        <h4>
        共搜到<span style="color: red; font-weight: bold;">
                <%=hitsList.size()%></span>条结果
        </h4>
```

```
            <%
                if (hitsList.size() > 0) {
                    Iterator<FileModel> iter = hitsList.iterator();
                    FileModel fm;
                    while (iter.hasNext()) {
                        fm = iter.next();
            %>
            <div class="item">
                <div class="itemtop">
                    <h4>
                        <img alt="pdf" src="images/<%=fm.getTitle()
                            .split("\\.")[1]%>.png"  class="doclogo">
                        <%=fm.getTitle().split("\\.")[0]%>
                    </h4>
                    <h3>
                        <a href="FileDownloadServlet?filename=
                        <%=regexHtml.delHtmlTag(fm.getTitle())%>">单击下载</a>
                    </h3>
                    </div>
                    <div class="itembuttom">
                        <p><%=fm.getContent().length() > 210 ?
                             fm.getContent().substring(0, 210):
                             fm.getContent()%>...</p>
                    </div>
                    <hr class="itemline">
                </div>
            <%
                }
                }
            %>
            </div>
            <div class="footer">
                <p>《Elasticsearch 入门与实战》之 Lucene 项目案例</p>
                <p>&copy;2016 清华大学出版社</p>
            </div>
    </body>
    </html>
```

搜索结果页面效果如图 3-13 所示。



图 3-13　文件检索查询结果页面

3.9　本章小结

本章通过 Lucene 文件检索项目，整合了 Lucene、Java Web、HTML、CSS，希望读者能够理清思路，步步为营，在实现这个小项目的过程中锻炼动手能力，提高编程技巧，加深对 Lucene 的理解。

第4章

从 Lucene 到 Elasticsearch

本章学习要点：

- ✱ Elasticsearch 与 Lucene 的关系
- ✱ Elasticsearch 架构
- ✱ Elasticsearch 的优点
- ✱ Elasticsearch 应用案例
- ✱ Elasticsearch 的核心概念
- ✱ Elasticsearch 和关系型数据库对比
- ✱ 安装和配置 Elasticsearch
- ✱ 安装和配置 IK 分词插件
- ✱ 安装和配置 Kibana

4.1 Elasticsearch 概述

4.1.1 诞生过程

Elasticsearch 是一个基于 Lucene 的搜索服务器，采用 Java 语言编写，使用 Lucene 构建索引、提供搜索功能，并作为 Apache 许可条款下的开放源码发布，是当前流行的企业级搜索引擎。我们知道，Lucene 提供的功能已经很强大了，但是 Lucene 只是一个由 Java 语言编写的库，对不使用 Java 语言的开发人员并不友好，Elasticsearch 在 Lucene 的基础上做了更多的改进，提供了多种语言接口。如图 4-1 所示，Lucene 之于 Elasticsearch 堪比发动机之于汽车，Elasticsearch 底层使用的仍然是 Lucene 的 API，Lucene 专注于底层搜索的建设，Elasticsearch 专注于企业应用。

Elasticsearch 的目标是让全文搜索变得简单，开发者可以通过简单明了的 RESTFul API 轻松地实现搜索功能，而不必去面对 Lucene 的复杂性。我们可以通过下面一段阅读材料了解 Elasticsearch 的诞生过程，以便更好地认识和定位 Elasticsearch。

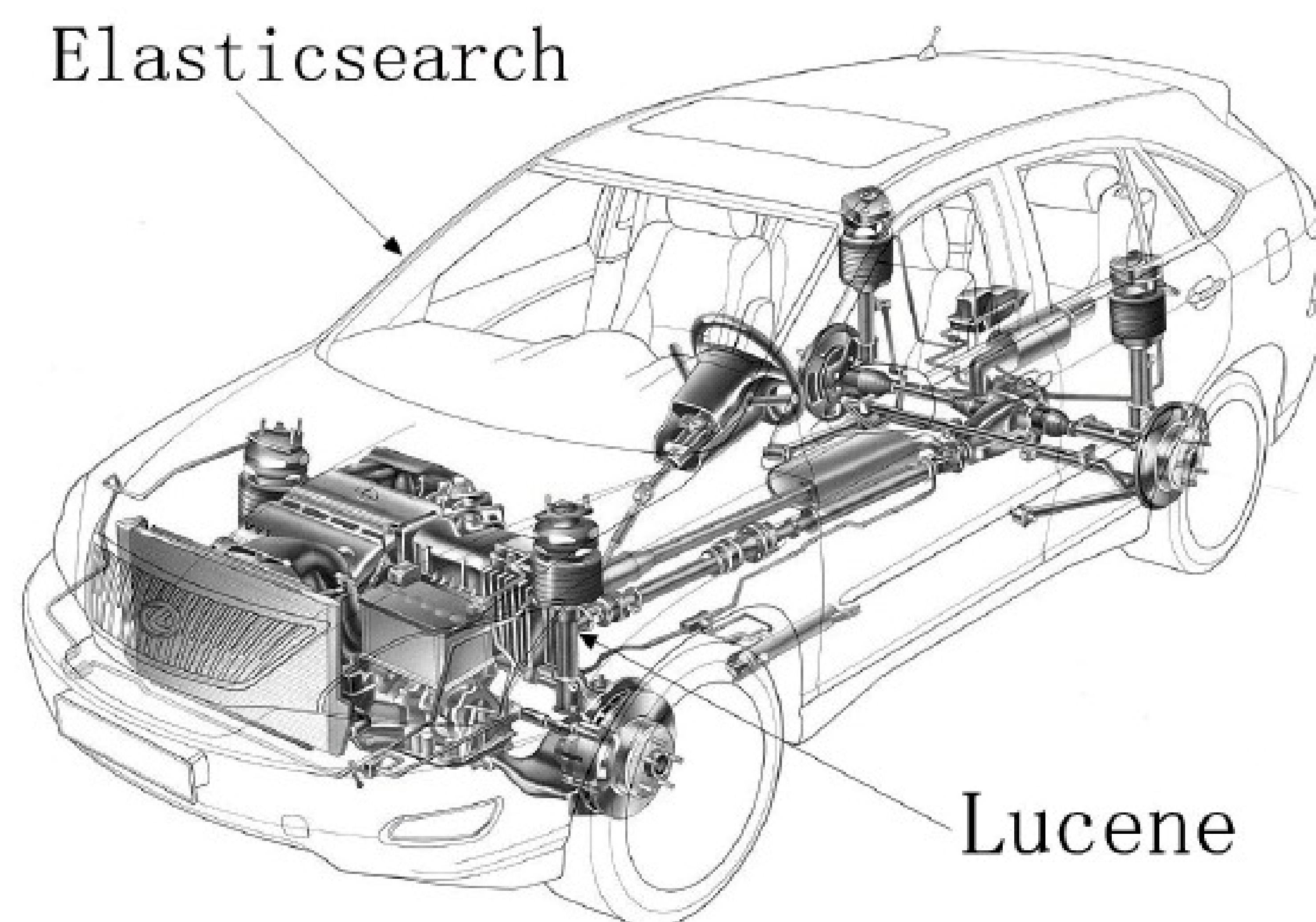


图 4-1 Elasticsearch 和 Lucene 之间的关系

待业工程师 Shay Banon 想为在伦敦学习做一名厨师的妻子开发一个方便搜索菜谱的应用而接触到 Lucene，Shay 在使用 Lucene 构建搜索的过程中遇到很多问题，包括大量重复性的工作，Shay 便在 Lucene 的基础上不断进行抽象和优化以便 Java 程序嵌入搜索更加方便，经过一段时间的打磨诞生了他的第一个开源作品“Compass（中文为指南针的意思）”。之后，Shay 找到了一份面对高性能分布式开发环境的新工作，在工作中他渐渐发现越来越需要一个易用的、高性能、实时、分布式搜索服务，Shay 在思考第三版的 Compass 时候意识到非常有必要重构 Compass 的大部分功能来实现“稳定搜索解决方案”的目标，于是 Compass 从一个库被打造成一个具有分布式功能、基于 JSON 和 HTTP 接口、适用非 Java 语言的独立 Server。2010 年 2 月，Shay Banon 发布了 Elasticsearch 的第一个版本。

如今 Elasticsearch 已经作为一家公司（Elastic 公司）进行运作，定位为数据搜索和分析平台，并在 2014 年 6 月获得 7000 万美元的融资，累积融资过亿美元。Shay Banon 在接受访谈时称，创建公司要做的第一件事就是确保所有流行的语言和框架都有正式的客户端驱动，现在 Elasticsearch 已经可以与 Java、Ruby、Python、PHP、Perl、.NET 等多种客户端集成。除此之外，Elasticsearch 也可以与 Hadoop、Spark 等大数据分析平台进行集成，功能十分强大。

基于 Elasticsearch 衍生出来了一系列开源软件，统称为 Elastic Stack，主要包括分布式搜索引擎 Elasticsearch、日志采集与解析工具 Logstash、可视化分析平台 Kibana、数据采集工具 Beats 家族等。在没有引入 Beats 之前，Elasticsearch、Logstash、Kibana 三者简称 ELK Stack，是非常流行的集中式日志解决方案。Logstash 既可以作为日志搜集器又能解析日志，但是 Logstash 会消耗较多的 CPU 和内存资源，容易造成服务器性能下降。后来 Elastic 公司推出了 Beats 家族，在数据收集方面使用 Beats 取代 Logstash，解决了 Logstash 在各服务器节点上占用系统资源高的问题。相比 Logstash，Beats 所占系统的 CPU 和内存几乎可以忽略不计，另外，Beats 和 Logstash 之间支持 SSL/TLS 加密传输以及客户端和服务器双向认证，保证了通信安全。Beats 家族的 5 个成员简介如下：

- Filebeat 轻量级的日志采集器，可用于收集文件数据。
- Metricbeat 5.0 版本之前名为 Topbeat，搜集系统、进程和文件系统级别的 CPU 和内存使用情况等数据。

- Packetbeat　收集网络流数据，可以实时监控系统应用和服务，可以将延迟时间、错误、响应时间、SLA 性能等信息发送到 Logstash 或 Elasticsearch。
- Winlogbeat　搜集 Windows 事件日志数据。
- Heartbeat　监控服务器运行状态。

为了避免版本混乱，从 5.0 版本开始，Elastic 公司将各组件的版本号统一。使用时，各组件版本号应该一致，Elastic Stack 的版本号为 X.Y.Z 的形式，例如 5.2.3。其中，各组件的 Z 可以不相同，但 X、Y 必须一致。

4.1.2　流行度分析

DB-Engines（官网：http://db-engines.com/）是一家收集和统计数据库管理系统信息的机构，不仅包含传统关系型数据库，对 NoSQL 领域也非常关注。在该网站可以查看各种 DBMS 的特点、流行程度等信息，同时也可以选择其他 DBMS 做对比。DB-Engines Ranking 是根据 DBMS 的流行程度做的排名，根据统计信息每月更新一次。如图 4-2 所示是 2017 年 8 月份搜索引擎类的流行度排名，Elasticsearch 排名第一，流行度超过了 Solr。如图 4-3 所示是各搜索引擎流行度趋势图，Elasticsearch 一直保持稳健的增长趋势。

16 systems in ranking, August 2017

Rank Aug 2017	Rank Jul 2017	Rank Aug 2016	DBMS	Database Model	Score Aug 2017	Score Jul 2017	Score Aug 2016
1.	1.	1.	Elasticsearch	Search engine	117.65	+1.67	+25.16
2.	2.	2.	Solr	Search engine	66.96	+0.93	+1.18
3.	3.	3.	Splunk	Search engine	61.46	+1.17	+12.56
4.	4.	4.	MarkLogic	Multi-model	12.50	+0.07	+2.46
5.	5.	5.	Sphinx	Search engine	6.15	-0.28	-1.87
6.	6.	↑8.	Microsoft Azure Search	Search engine	3.29	+0.02	+1.69
7.	7.	↓6.	Google Search Appliance	Search engine	2.89	-0.08	-0.16
8.	8.		Algolia	Search engine	2.61	+0.11	
9.	9.	↓7.	Amazon CloudSearch	Search engine	2.30	-0.01	-0.03
10.	10.	↑11.	CrateDB	Multi-model	0.85	-0.05	+0.63
11.	11.	↓9.	Xapian	Search engine	0.58	-0.01	+0.11
12.	12.	↑13.	SearchBlox	Search engine	0.26	-0.02	+0.14
13.	↑14.	↑15.	Exorbyte	Search engine	0.15	-0.02	+0.15
14.	↓13.	↓10.	Indica	Search engine	0.09	-0.09	-0.24
15.	↑16.	15.	searchxml	Multi-model	0.03	+0.03	+0.03
16.	↓15.	↓14.	DBSight	Search engine	0.02	-0.01	+0.01

图 4-2　2017 年 8 月份搜索引擎排名

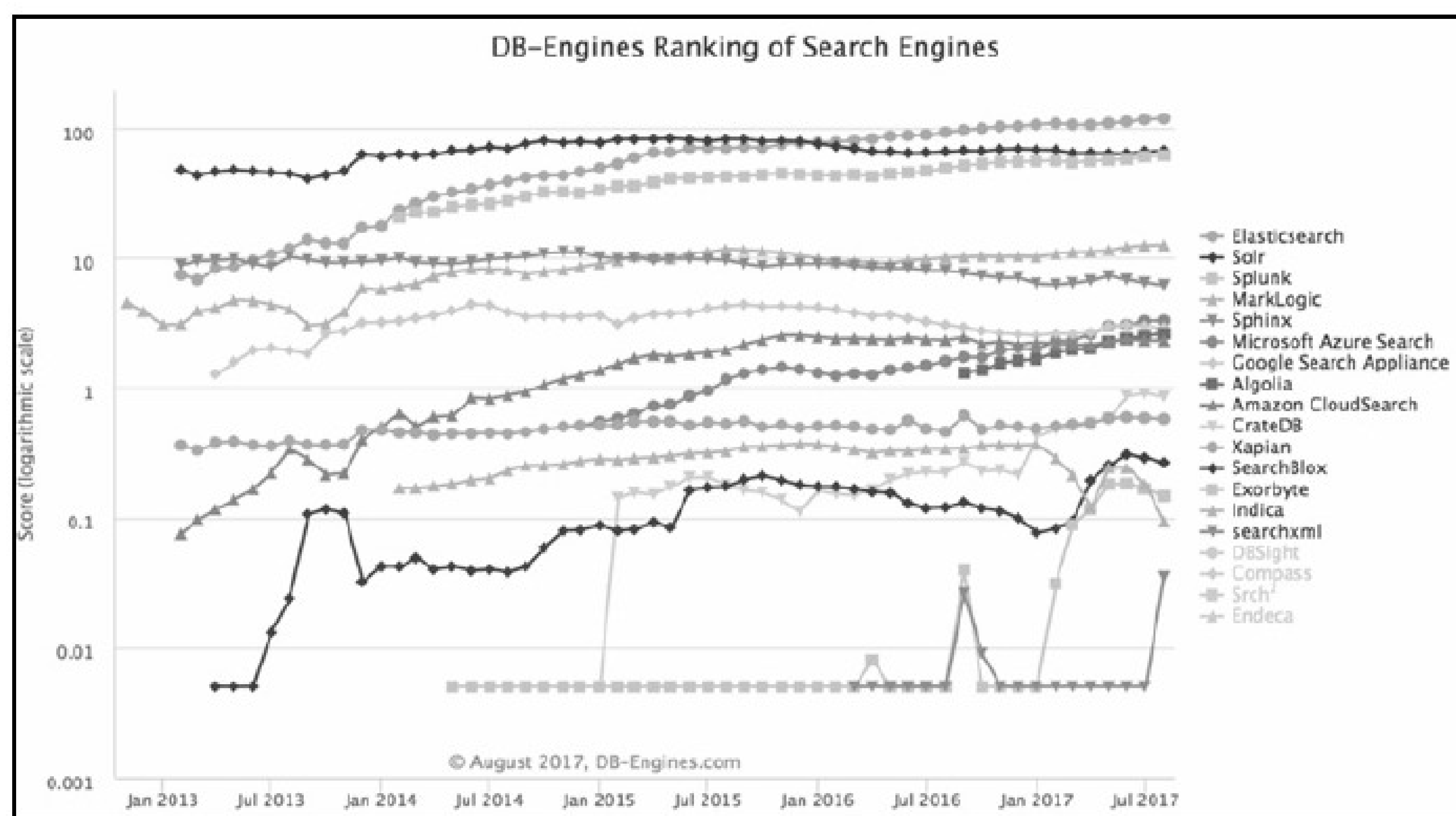


图 4-3　2017 年 8 月搜索引擎流行度趋势图

4.1.3　架构解读

如图 4-4 所示是 Elasticsearch 的整体架构，下面由下往上逐层介绍。

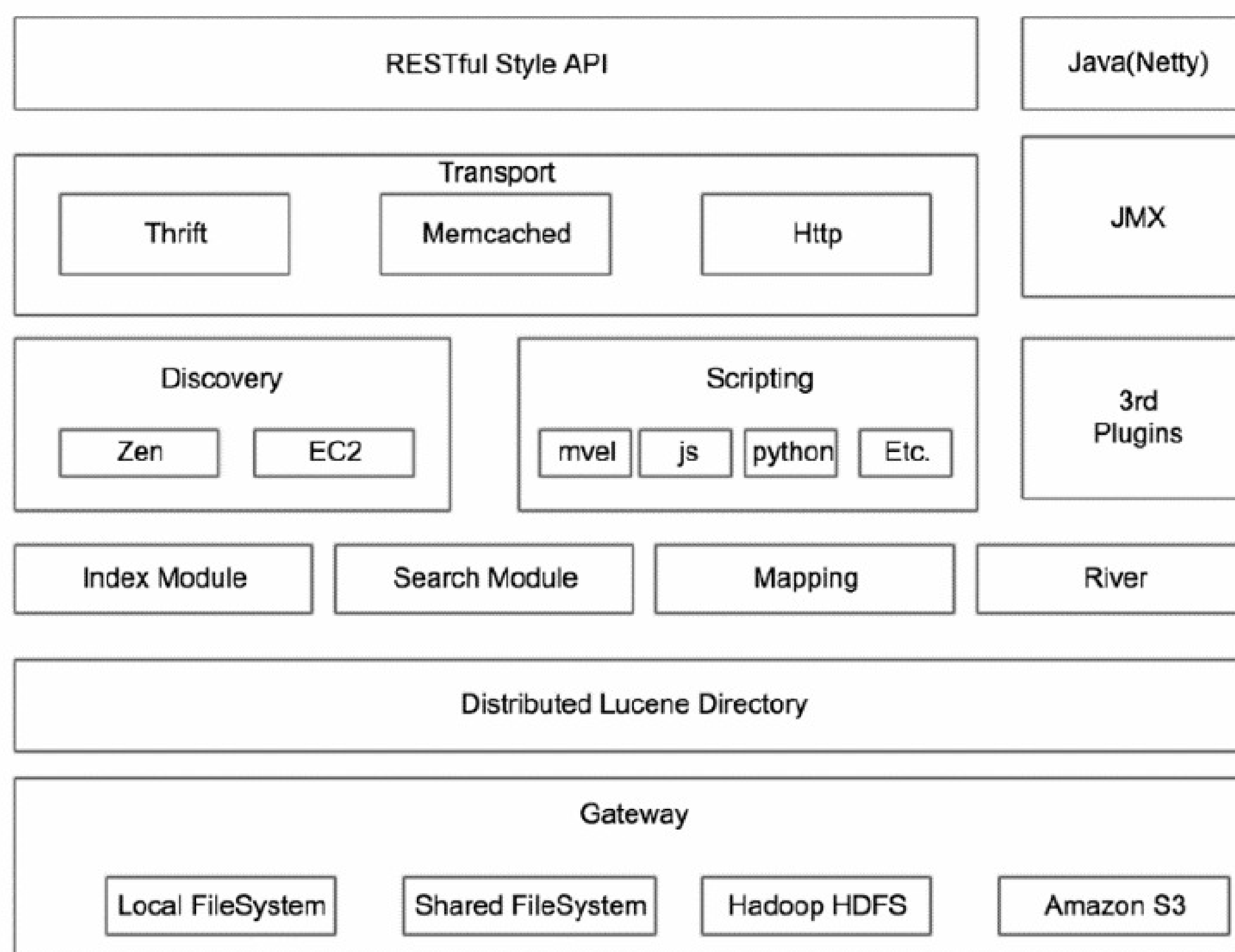


图 4-4　Elasticsearch 整体架构

Gateway 是 Elasticsearch 用来存储索引的文件系统，支持多种文件类型，Local FileSystem 是存储在本地的文件系统，Shared FileSystem 是共享存储，也可以使用 Hadoop 的 HDFS 分布式存储，也可以存储在 Amazon S3 云服务上。

Gateway 的上层是一个分布式的 Lucene 框架，Elasticsearch 的底层 API 是由 Lucene 提供的，每一个 Elasticsearch 节点上都有一个 Lucene 引擎的支持。

Lucene 之上是 Elasticsearch 的模块，包括索引模块、搜索模块、映射解析模块等。River 相当于第三方插件，用来导入第三方数据源，在 2.X 之后已经不再使用。

Elasticsearch 模块之上是 Discovery、Scripting 和第三方插件。Discovery 是 Elasticsearch 的节点发现模块，不同机器上的 Elasticsearch 节点要组成集群需要进行消息通信，集群内部需要选举 master 节点，这些工作都是由 Discovery 模块完成的。Scripting 用来支持 JavaScript、Python 等多种语言，可以在查询语句中嵌入，使用 Script 语句性能稍低。Elasticsearch 也支持多种第三方插件。

再上层是 Elasticsearch 的传输模块和 JMX。传输模块支持 Thrift、Memcached、HTTP，默认使用 HTTP 传输。JMX 是 Java 的管理框架，用来管理 Elasticsearch 应用。

最上层是 Elasticsearch 提供给用户的接口，可以通过 RESTful API 和 Elasticsearch 集群进行交互。

4.1.4　优点

如图 4-5 所示来源于百度大数据部 2015 年做的题为《百度 Elasticsearch 实践》的分享，介绍了 Elasticsearch 的特点和优点，分析如下。

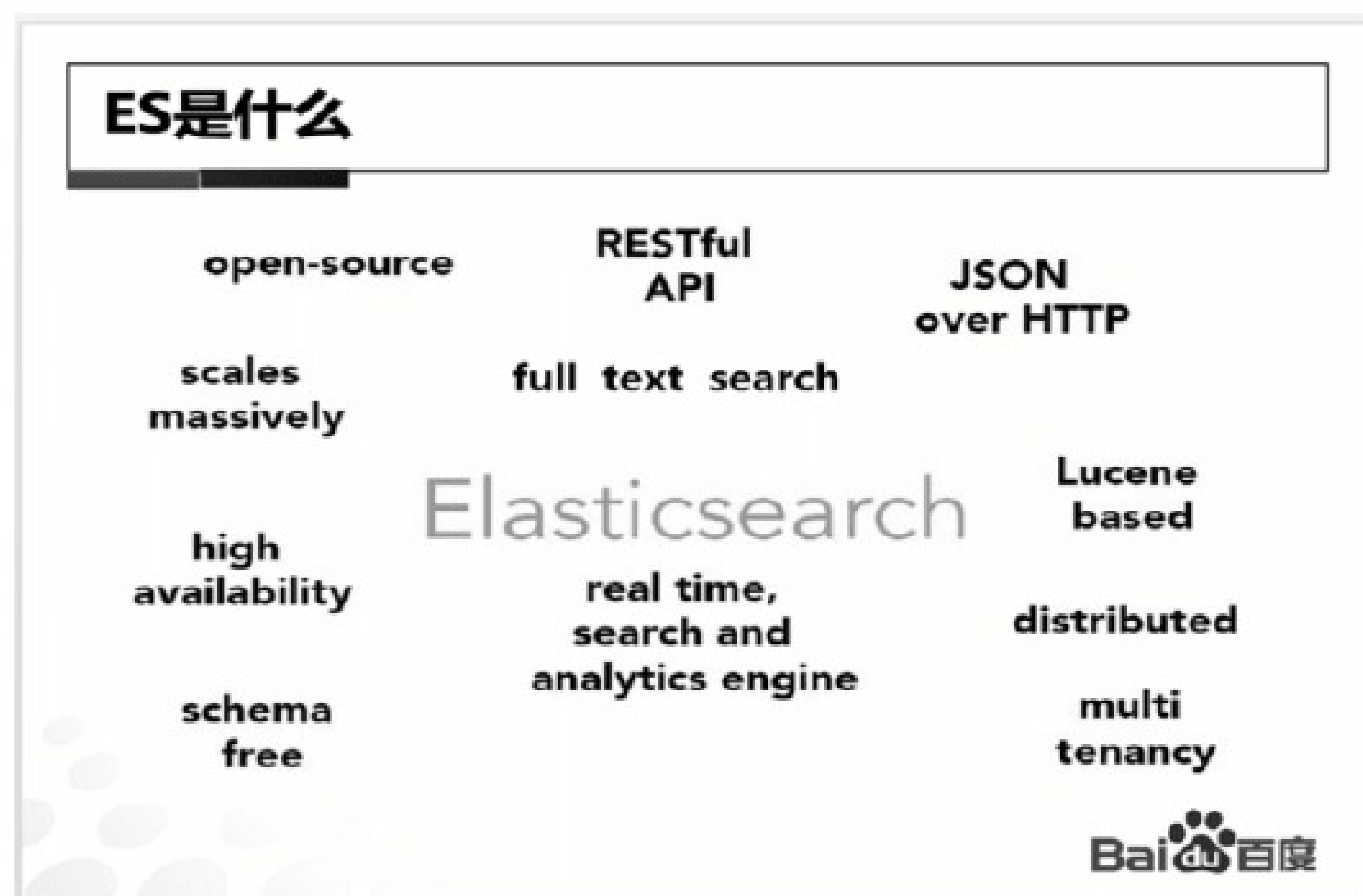


图 4-5　图片来源于《百度 Elasticsearch 实践》

- 分布式：Elasticsearch 横向扩展非常灵活，当数据规模比较小的时候可以使用小规模的集群。随着数据的增长，需要更大的容量和更高的性能，此时只需增加更多的节点，Elasticsearch 的自动发现机制会识别新增的节点并重新平衡分配数据。
- 全文检索：Apache Lucene 是一个用 Java 编写的高性能的功能齐全的信息检索库，Elasticsearch 在后台使用 Lucene 来提供最强大的全文检索，提供任何开源产品的能力。自带多语言支持、强大的查询语言、地理位置支持、上下文感知的建议、自动完成和搜索片段。
- 近实时搜索和分析：数据从进入 Elasticsearch，可达到近实时搜索。除了搜索，Elasticsearch 也可以进行聚合分析操作。
- 高可用：高可用主要体现在容错机制上，Elasticsearch 集群会自动发现新的或失败的节点，重组和重新平衡数据，确保数据是安全的和可访问的。
- 模式自由：Elasticsearch 的动态 mapping 机制可以自动检测数据的结构和类型，创建索引，并使数据可搜索。
- RESTful API：Elasticsearch 是 API 驱动。几乎任何操作都可以用一个简单的 RESTful API 使用 JSON 基于 HTTP 请求来实现，客户端也可使用多种编程语言。

4.1.5　应用场景

Elaticsearch 应用场景可分为以下几类。

1. 站内搜索

Elaticsearch 在站内搜索中应用十分广泛，大部分网站尤其是网页信息量较大的网站，都会有站内全文检索这一功能，目的是为了方便用户快速检索信息。如图 4-6 所示是一个图书馆站内搜索的例子。

2. NoSQL 数据库

Elasticsearch 在读写性能上优于 MongoDB，同时也支持地理位置查询。

3. 日志分析

日志分析由实时日志分析平台 ELK（ELK 由 Elasticsearch、Logstash 和 Kiabana 三个开源工具组成）完成，能够对日志进行集中的收集、存储、搜索、分析、监控以及可视化。



图 4-6　图书馆站内搜索例子

国内外使用 Elasticsearch 的知名企业有：Facebook、Wikipedia、GitHub、Quora、Facebook、Sony、Mozilla、Adobe、NETFLIX、SoundCloud、Foursquare、LinkedIn、Ubnt、百度、新浪等。

- 维基百科使用 Elasticsearch 来进行全文搜索并高亮显示关键词，以及提供 search-as-you-type、did-you-mean 等搜索建议功能。
- 英国卫报使用 Elasticsearch 来处理访客日志，以便能将公众对不同文章的反应实时地反馈给各位编辑。
- GitHub 使用 Elasticsearch 来检索超过 1300 亿行代码。
- SoundCloud 是一家德国网站，提供音乐分享社区服务，使用 Elasticsearch 来为 1.8 亿用户提供即时精准的音乐搜索服务。
- Mozilla 公司以火狐著名，它目前使用 WarOnOrange 这个项目来进行单元或功能测试，测试的结果以 JSON 的方式索引到 Elasticsearch 中，开发人员可以非常方便地查找 bug。Socorro 是 Mozilla 公司的程序崩溃报告系统，一有错误信息就插入到 Hbase 和 Postgres 中，然后从 Hbase 中读取数据索引到 Elasticsearch 中，方便查找。
- Sony 公司使用 Elasticsearch 作为信息搜索引擎。
- 百度自 2013 年开始使用 Elasticsearch，覆盖 Casio、云分析、网盟、预测、文库、直达号、百度钱包、百度糯米等多个业务线，使用上百台服务器每天处理 TB 量级的数据。

如图 4-7 所示来源于 2016 年第五届 Elasticsearch 开发者大会上百度大数据部做的题为《百度对 Elasticsearch 的优化改进》的分享，介绍了 Elasticsearch 在百度的使用情况。

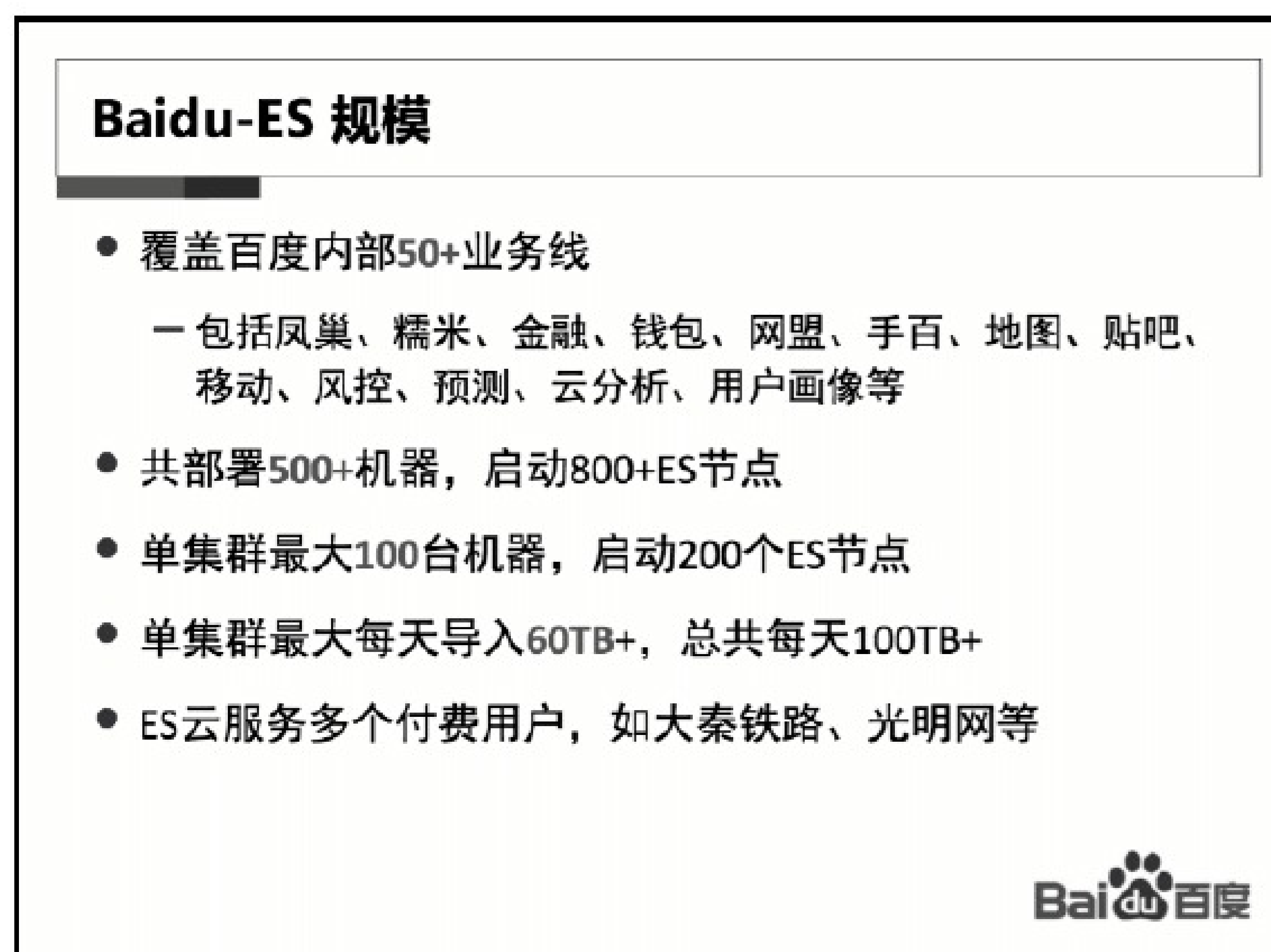


图 4-7　图片来源于《百度对 Elasticsearch 的优化改进》

更多的用户案例可以到 Elasticsearch 官网查看：https://www.elastic.co/use-cases。

4.1.6　核心概念

下面介绍 Elasticsearch 的一些核心概念，包括集群、节点、索引、类型、 文档、分片和副本。

1. 集群

一个或多个安装 Elasticsearch 的服务器节点组织在一起就是集群，它们共同持有你整个的数据，并一起提供索引和搜索功能。一个集群由一个唯一的名字标识，称为 cluster name，集群名称默认是“elasticsearch”。集群名称非常重要，就像一个组织的名称，具有相同集群名称的节点才会组成一个集群。集群名称可以在配置文件中指定。

2. 节点

一个节点是你集群中的一个服务器，作为集群的一部分，它存储你的数据，参与集群的索引和搜索功能。

一个节点可以通过配置集群名称的方式来加入一个指定的集群。默认情况下，每个节点都会被安排加入到一个集群名称为“elasticsearch”的集群中，这意味着如果你在你的网络中启动了若干个节点，并假定它们能够相互发现彼此，它们将会自动地形成并加入到一个叫做“elasticsearch”的集群中。但是有的时候这种机制并不可靠，会发生脑裂现象，往往不如在每一个节点上配置节点的名字在启动时进行被动发现来的安全稳定。

3. 索引

一个索引就是一个拥有几分相似特征的文档的集合，索引的数据结构仍然是倒排索引。比如说，你可以有一个客户数据的索引，另一个产品目录的索引，还有一个订单数据的索引。一个索引由一个名字来标识（必须全部是小写字母的），并且当我们要对这个索引中的文档进行索引、搜索、更新和删除的时候，都要使用到这个名字。在一个集群中，可以定义任意多的索引。索引做动词来讲的时候表示索引数据和对数据进行索引操作。

4. 类型

在一个索引中，你可以定义一种或多种类型。一个类型是你的索引的一个逻辑上的分类或分区，其语义完全由你来定。通常，会为具有一组共同字段的文档定义一个类型。

5. 文档

一个文档是一个可被索引的基础信息单元。比如，你可以拥有某一个客户的文档，某一个产品的一个文档。当然，也可以拥有某个订单的一个文档。文档都是 JSON 格式。

6. 分片

一个索引可以存储超出单个节点硬件限制的大量数据。比如，一个具有 10 亿文档的索引占据 1TB 的磁盘空间，而任一节点都没有这样大的磁盘空间；或者单个节点处理搜索请求，响应太慢。为了解决这个问题，Elasticsearch 提供了将索引划分成多份的能力，这些份就叫作分片。当你创建一个索引的时候，可以指定想要的分片的数量，每个分片本身也是一个功能完善并且独立的“索引”，这个“索引”可以被放置到集群中的任何节点上。

分片之所以重要，主要有以下两方面的原因：

（1）允许你水平分割/扩展你的内容容量。

（2）允许你在分片（潜在地，位于多个节点上）上进行分布式的、并行的操作，进而提高性能和吞吐量，至于一个分片怎样分布，它的文档怎样聚合回搜索请求，完全由 Elasticsearch 管理，对于用户来说，这些都是透明的。

7. 副本

在一个网络/云的环境里，失败随时都可能发生，在某个分片/节点不知怎么的就处于离线状态，或者由于某种原因消失了，这种情况下，有一个故障转移机制非常有用，并且也是强烈推荐的。为此，Elasticsearch 允许你创建分片的一份或多份拷贝，这些拷贝叫作复制分片，或者直接叫副本。

副本之所以重要，有以下两个主要原因：

（1）在分片/节点失败的情况下，保证高可用性。因为这个原因，复制分片不与主分片置于同一节点上，这一点非常重要。

（2）扩展你的搜索量/吞吐量，因为搜索可以在所有的副本上并行运行。

总之，每个索引可以被分成多个分片。一个索引可以有一至多个副本。一旦有了副本，每个索引就有了主分片（作为复制源的原来的分片）和副本分片（主分片的拷贝）之别。分片和副本的数量可以在索引创建的时候指定。在索引创建之后，可以在任何时候动态地改变副本的数量，但是事后不能改变分片的数量。

4.1.7　对比 RDMS

Elasticsearch 可以看成一个数据库，只是和关系型数据库比起来数据格式和功能不一样而已。如果是初次接触 Elasticsearch，对一些概念比较陌生的话，可以参考表 4-1 对比 RDMS 理解 Elasticsearch 的一些术语。

表4-1　Elasticsearch和RDMS对比

RDMS	Elasticsearch
数据库（database）	索引（index）
表（table）	类型（type）
行（row）	文档（document）
列（column）	字段（field）
表结构（Schema）	映射（Mapping）
索引	全文索引
SQL	查询 DSL
SELECT * from tablename	GET http://......
UPDATE table SET	PUT http://......
DELETE	DELETE http://......

请注意，进入 Elasticsearch 的世界以后，我们讲到的某个索引下的某个类型的某个文档，和关系型数据库讲某个数据库中的某张表的某条记录是等价的。

4.1.8　文档结构

了解 JSON 这种数据格式的结构和特点非常有必要，因为文档是 Elasticsearch 中的基本单位，Elasticsearch 中的文档又都是用 JSON 来表示的，以后我们还要经常和 JSON 打交道。在开始 Elasticsearch 学习之前先来学习一下 JSON，对 JSON 已经很熟悉的读者可以跳过本节。

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，易于人阅读和编写，同时也易于机器解析和生成。它基于 JavaScript Programming Language，是 Standard ECMA-262 3rd Edition-December 1999 的一个子集。JSON 采用完全独立于语言的文本格式，但是也使用了类似于 C 语言家族的习惯（包括 C、C++、C#、Java、JavaScript、Perl、Python 等）。JSON 使用 JavaScript 语法来描述数据对象，但是 JSON 仍然独立于语言和平台。JSON 解析器和 JSON 库支持许多不同的编程语言，这些特性使 JSON 成为理想的数据交换语言。

JSON 主要有以下两种结构：

- “key/value”键值对结构。在不同的语言中，它被理解为对象（object）、纪录（record）、结构（struct）、字典（dictionary）、哈希表（hash table）、有键列表（keyed list）或者关联数组（associative array）。键/值对包括字段名称（在双引号中），后面写一个冒号，然后是值，比如："name"："Tom"，等价于 JavaScript 语句：name="Tom"。
- 数组结构。也称为值的有序列表（An ordered list of values）。在大部分语言中都被理解为数组（array）。

JSON 值的类型可以是数字（整型或浮点型）、字符串（在双引号之内）、布尔值（true 或 false）、数组（在方括号之内）、对象（在花括号之内）、null。

下面是 JSON 对象的一个例子：

```
{
    "name": "JSON 中国",
    "url": "http://www.json.org.cn",
    "page": 88,
    "isNonProfit": true,
    "address": {
        "street": "浙大路 38 号.",
        "city": "浙江杭州",
        "country": "中国"
    },
    "links": [
        {
            "name": "Google",
            "url": "http://www.google.com"
        },
        {
            "name": "Baidu",
            "url": "http://www.baidu.com"
        }
    ]
}
```

JSON 对象在花括号中书写，对象可以包含多个名称/值对，在上面的例子中，对象“name”“url”“page”“isNonProfit”的值都是简单类型，对象“address”包含 3 个嵌套对象，对象“links”包含 2 个对象数组。

和另一种数据交换常用文件格式 XML 相比，JSON 有诸多优点。首先 JSON 数据格式比较简单，易于读写，格式都是压缩的，占用带宽小，而 XML 文件庞大，文件格式复杂，传输占带宽比较大。其次，在解码难度上，JSON 更易于解析，XML 的解析得考虑父节点和子节点，让人头昏眼花，而 JSON 的解析难度几乎为零。此外，服务器端和客户端都需要花费大量代码来解析 XML，导致服务器端和客户端代码变得异常复杂且不易维护，而 JSON 格式则能直接为服务器端代码使用，大大简化了服务器端和客户端的代码开发量。

4.2　安装 Elasticsearch

Elasticsearch 的安装需要 Java 的支持，强烈建议读者在 Linux 环境下学习 Elasticsearch，Elasticsearch 对 Linux 操作系统的支持较好，在 Linux 下学习 Elasticsearch 能减少不必要的麻烦，生产环境中 Elasticsearch 集群一般也都部署在 Linux 服务器上。下面从安装 Java 开始，介绍 Elasticsearch 的安装步骤。

4.2.1 安装 Java

步骤 01 下载 Java。

运行 Elasticsearch 5.0 及以上的版本要求 Java 版本不低于 1.8，首先到 Oracle 官网下载 Java，访问 http://www.oracle.com/technetwork/java/javase/downloads/index.html，然后单击 Java Download 图标，之后会跳转到如图 4-8 所示的 JDK 下载列表。选中 Accept License Agreement，然后根据不同的系统选择相应的 java 安装包，这里下载 jdk-8u144-linux-x64.tar.gz。

Java SE Development Kit 8u144

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.
Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.89 MB	jdk-8u144-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.83 MB	jdk-8u144-linux-arm64-vfp-hflt.tar.gz
Linux x86	164.65 MB	jdk-8u144-linux-i586.rpm
Linux x86	179.44 MB	jdk-8u144-linux-i586.tar.gz
Linux x64	162.1 MB	jdk-8u144-linux-x64.rpm
Linux x64	176.92 MB	jdk-8u144-linux-x64.tar.gz
Mac OS X	226.6 MB	jdk-8u144-macosx-x64.dmg
Solaris SPARC 64-bit	139.87 MB	jdk-8u144-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.18 MB	jdk-8u144-solaris-sparcv9.tar.gz
Solaris x64	140.51 MB	jdk-8u144-solaris-x64.tar.Z
Solaris x64	96.99 MB	jdk-8u144-solaris-x64.tar.gz
Windows x86	190.94 MB	jdk-8u144-windows-i586.exe
Windows x64	197.78 MB	jdk-8u144-windows-x64.exe

图 4-8　Java SE 下载列表

步骤 02 解压 Java。

解压安装包：

```
tar -zxvf jdk-8u144-linux-x64.tar.gz
```

解压完以后会生成文件夹 jdk1.8.0_144，然后在/opt 目录下（可自定义）新建文件夹：

```
sudo mkdir  /opt/javahome
```

移动 java 文件到/opt/javahome：

```
sudo mv jdk1.8.0_144 /opt/javahome
```

步骤 03 设置环境变量。

编辑配置文件：

```
vim /etc/profile
```

在文件末尾添加以下内容：

```
export JAVA_HOME=/opt/javahome/jdk1.8.0_144
export PATH=$PATH:$JAVA_HOME/bin
export CLASSPATH=$JAVA_HOME/lib
export CLASSPATH=$CLASSPATH:$JAVA_HOME/jre/lib
```

使刚才的配置生效，执行命令：

```
source /etc/profile
```

步骤 04 测试 Java 是否安装成功。

执行命令：

```
java -version
```

如果返回如下的 Java 版本信息，说明安装成功。

```
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
```

4.2.2 下载 Elasticsearch

访问 Elastic 官网的软件下载地址：https://www.elastic.co/downloads/past-releases，下载页面如图 4-9 所示，在左边选择要下载的软件名称，右边选择软件版本。单击 Download 按钮，会跳转到如图 4-10 所示的安装包下载页，Linux 系统下载 tar 格式安装包。

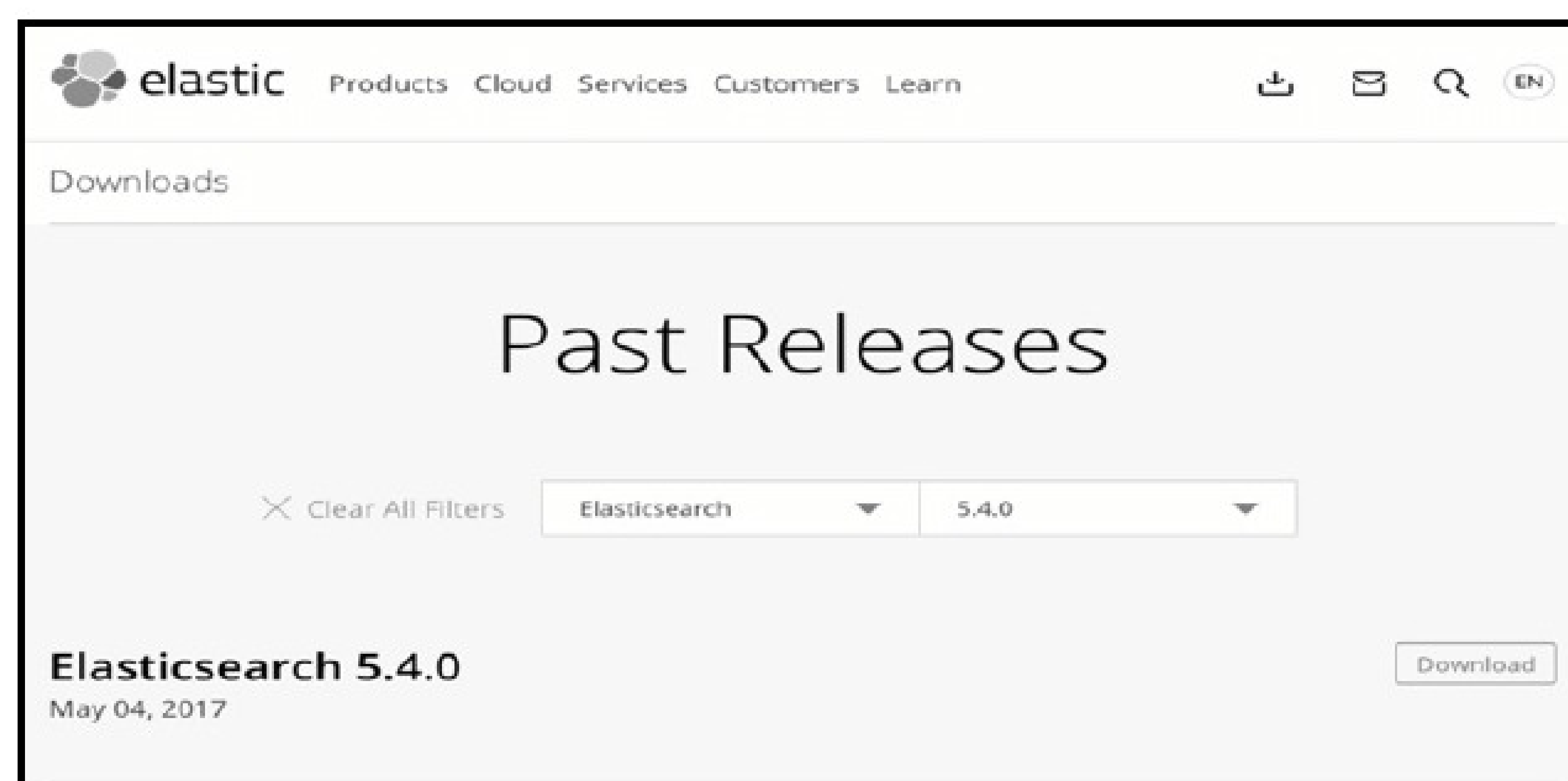


图 4-9　Elasticsearch 下载

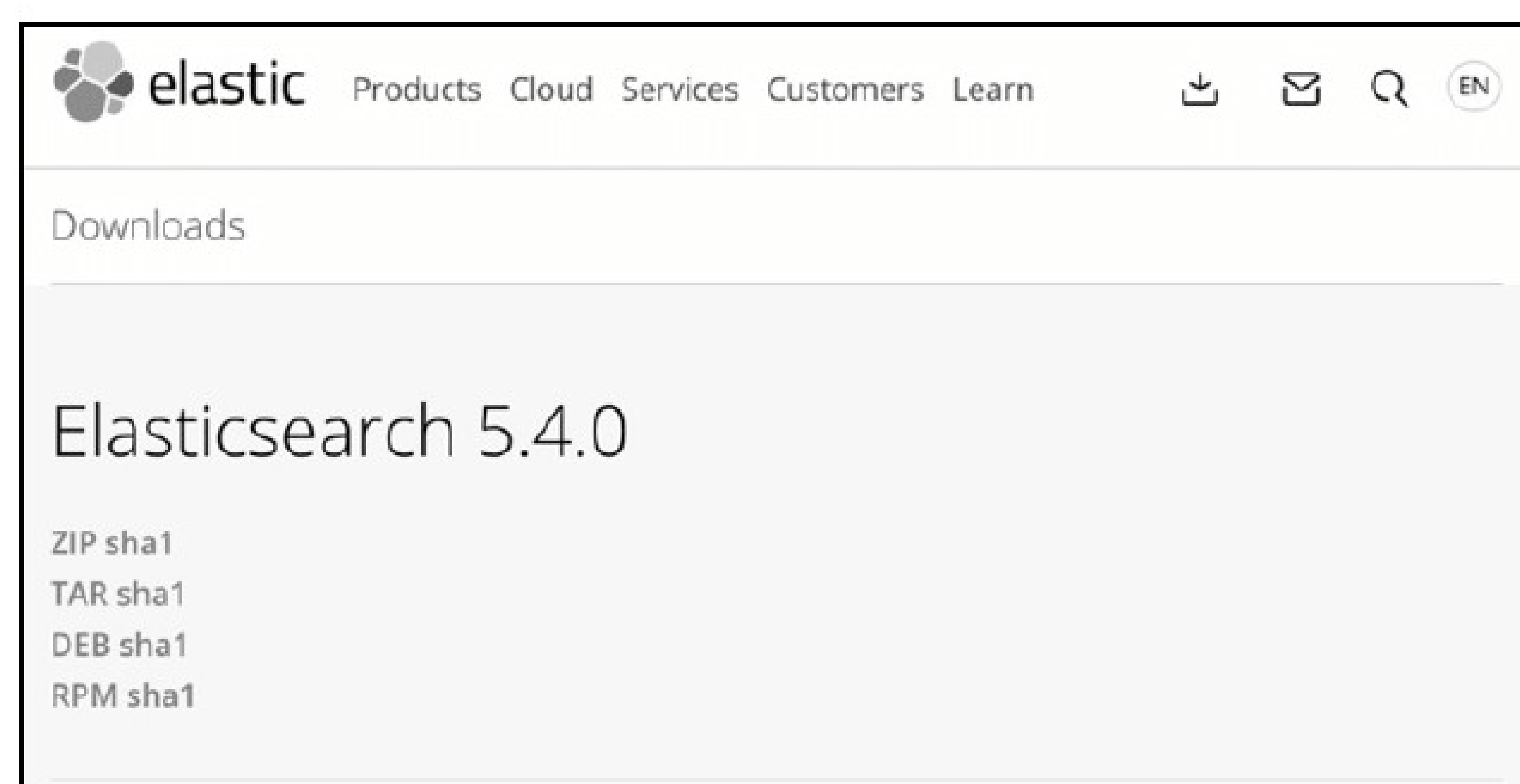


图 4-10　Elasticsearch 下载页面

4.2.3 启动 Elasticsearch

下载完成之后解压 tar 文件，解压之后不需要任何配置，切换到 elasticsearch-5.4.0 目录下，打开终端执行启动命令：./bin/elasticsearch，如果一切顺利，会收到如图 4-11 所示的输出信息。

```
1. java
Bee:elasticsearch-5.4.0 bee$ ./bin/elasticsearch
[2017-08-24T13:35:46,771][INFO ][o.e.n.Node               ] [] initializing ...
[2017-08-24T13:35:46,839][INFO ][o.e.e.NodeEnvironment    ] [CHo4z-e] using [1] data paths, mounts [[/ (/dev/disk1)]], net usable_space [25.5gb], net total_space [111.8gb], spins? [unknown], types [hfs]
[2017-08-24T13:35:46,839][INFO ][o.e.e.NodeEnvironment    ] [CHo4z-e] heap size [1.9gb], compressed ordinary object pointers [true]
[2017-08-24T13:35:46,908][INFO ][o.e.n.Node               ] node name [CHo4z-e] derived from node ID [CHo4z-eNRiiFmwyx3v4Nwg]; set [node.name] to override
[2017-08-24T13:35:46,908][INFO ][o.e.n.Node               ] version[5.4.0], pid[4374], build[780f8c4/2017-04-28T17:43:27.229Z], OS[Mac OS X/10.10.3/x86_64], JVM[Oracle Corporation/Java HotSpot(TM) 64-Bit Server VM/1.8.0_121/25.121-b13]
[2017-08-24T13:35:47,930][INFO ][o.e.p.PluginsService     ] [CHo4z-e] loaded module [aggs-matrix-stats]
[2017-08-24T13:35:47,931][INFO ][o.e.p.PluginsService     ] [CHo4z-e] loaded module [ingest-common]
[2017-08-24T13:35:47,931][INFO ][o.e.p.PluginsService     ] [CHo4z-e] loaded module [lang-expression]
[2017-08-24T13:35:47,931][INFO ][o.e.p.PluginsService     ] [CHo4z-e] loaded module [lang-groovy]
[2017-08-24T13:35:47,931][INFO ][o.e.p.PluginsService     ] [CHo4z-e] loaded module [lang-mustache]
[2017-08-24T13:35:47,931][INFO ][o.e.p.PluginsService     ] [CHo4z-e] loaded module [lang-painless]
[2017-08-24T13:35:47,931][INFO ][o.e.p.PluginsService     ] [CHo4z-e] loaded module [percolator]
[2017-08-24T13:35:47,932][INFO ][o.e.p.PluginsService     ] [CHo4z-e] loaded module [reindex]
[2017-08-24T13:35:47,932][INFO ][o.e.p.PluginsService     ] [CHo4z-e] loaded module [transport-netty3]
[2017-08-24T13:35:47,932][INFO ][o.e.p.PluginsService     ] [CHo4z-e] loaded module [transport-netty4]
[2017-08-24T13:35:47,932][INFO ][o.e.p.PluginsService     ] [CHo4z-e] loaded plugin [analysis-ik]
[2017-08-24T13:35:49,525][INFO ][o.e.d.DiscoveryModule    ] [CHo4z-e] using discovery type [zen]
[2017-08-24T13:35:50,155][INFO ][o.e.n.Node               ] initialized
[2017-08-24T13:35:50,156][INFO ][o.e.n.Node               ] [CHo4z-e] starting ...
[2017-08-24T13:35:50,306][INFO ][o.e.t.TransportService   ] [CHo4z-e] publish_address {127.0.0.1:9300}, bound_addresses {[fe80::1]:9300}, {[::1]:9300}, {127.0.0.1:9300}
[2017-08-24T13:35:53,448][INFO ][o.e.c.s.ClusterService   ] [CHo4z-e] new_master {CHo4z-e}{CHo4z-eNRiiFmwyx3v4Nwg
```

图 4-11　终端启动 Elasticsearch

Elasticsearch 默认的 HTTP 端口是 9200，TCP 端口是 9300，执行下面的命令访问 9200 端口：curl localhost:9200，输出结果如图 4-12 所示。

也可以如图 4-13 所示在浏览器中直接访问 9200 端口，返回结果和命令行访问是一样的，都是一个 JSON 对象。

```
2. bash
Bee:~ bee$ curl localhost:9200
{
  "name" : "CHo4z-e",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "gBcawmhrQkaKK-HFh35mEw",
  "version" : {
    "number" : "5.4.0",
    "build_hash" : "780f8c4",
    "build_date" : "2017-04-28T17:43:27.229Z",
    "build_snapshot" : false,
    "lucene_version" : "6.5.0"
  },
  "tagline" : "You Know, for Search"
}
Bee:~ bee$
```

图 4-12　终端访问 Elasticsearch 9200 端口

```
localhost:9200
{
  "name" : "CHo4z-e",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "gBcawmhrQkaKK-HFh35mEw",
  "version" : {
    "number" : "5.4.0",
    "build_hash" : "780f8c4",
    "build_date" : "2017-04-28T17:43:27.229Z",
    "build_snapshot" : false,
    "lucene_version" : "6.5.0"
  },
  "tagline" : "You Know, for Search"
}
```

图 4-13　浏览器中访问 Elasticsearch 9200 端口

通过返回信息可以知道，已经成功启动了一个 Elasticsearch 节点，节点的名称为“CHo4z-e”，集群名称是默认的“elasticsearch”，还可以看到 Elasticsearch 的版本是 5.4.0，Lucene 版本 6.5.0，该版本的发布日期是 2017 年 4 月 28 日。

Elasticsearch 的根目录下各文件的作用见表 4-2。

表4-2　Elasticsearch安装目录文件说明

文件（夹）	功　能
bin	可执行文件目录
config	配置文件目录
data	数据存储目录

（续表）

文件（夹）	功　能
lib	第三方依赖库
logs	输出日志目录
modules	依赖模块目录
plugins	插件目录
LICENSE.txt	LICENSE 声明文件
NOTICE.txt	版权声明文件
README.textile	Elasticsearch 介绍信息

4.2.4　后台运行 Elasticsearch

控制台输出的方式启动 Elasticsearch 适用于测试阶段或者开发阶段，在生产环境中，Elasticsearch 应当作为系统服务在后台运行。后台启动 Elasticsearch 的命令如下：

```
./bin/elasticsearch  -d
```

加上-d 参数以后 Elasticsearch 会在后台启动，不会在终端中打印集群启动的相关信息。因为 Elasticsearch 是运行在 JVM 之上，可以使用 jps（JavaVirtual Machine Process Status Tool 的缩写，是 JDK 提供的一个查看当前 Java 进程的小工具）命令查看 Elasticsearch 是否启动成功。如图 4-14 所示，在启动 Elasticsearch 之前，使用 jps 命令查看当前 Java 进程是没有 Elasticsearch 进程的，执行后台启动命令之后再次查看就会多出一个 Elasticsearch 进程，前面的数字代表 Elasticsearch 的进程 ID。

```
1. bash
Bee:elasticsearch-5.4.0 bee$ jps
10184 Jps
Bee:elasticsearch-5.4.0 bee$ ./bin/elasticsearch -d
Bee:elasticsearch-5.4.0 bee$ jps
10211 Jps
10207 Elasticsearch
Bee:elasticsearch-5.4.0 bee$ ps aux|grep elasticsearch
bee              10207   1.6 20.0  6197540 1679272 s000  S     6:05下午   0:20.91
 /usr/bin/java -Xms2g -Xmx2g -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyF
raction=75 -XX:+UseCMSInitiatingOccupancyOnly -XX:+DisableExplicitGC -XX:+Always
PreTouch -server -Xss1m -Djava.awt.headless=true -Dfile.encoding=UTF-8 -Djna.nos
ys=true -Djdk.io.permissionsUseCanonicalPath=true -Dio.netty.noUnsafe=true -Dio.
netty.noKeySetOptimization=true -Dio.netty.recycler.maxCapacityPerThread=0 -Dlog
4j.shutdownHookEnabled=false -Dlog4j2.disable.jmx=true -Dlog4j.skipJansi=true -X
X:+HeapDumpOnOutOfMemoryError -Des.path.home=/Users/bee/Desktop/es5csdn/tools/el
asticsearch-5.4.0 -cp /Users/bee/Desktop/es5csdn/tools/elasticsearch-5.4.0/lib/*
 org.elasticsearch.bootstrap.Elasticsearch -d
bee              10221   0.0  0.0  2432772     652 s000  S+    6:06下午   0:00.00
grep elasticsearch
Bee:elasticsearch-5.4.0 bee$ kill 10207
Bee:elasticsearch-5.4.0 bee$ jps
10230 Jps
Bee:elasticsearch-5.4.0 bee$
```

图 4-14　查找并关闭 Elasticsearch 进程

4.2.5　关闭 Elasticsearch

需要对服务器进行重启或者关机时，首先要关闭正在运行的 Elasticsearch。目前有两种方式关闭 Elasticsearch：如果 Elasticsearch 是控制台方式输出运行，可以在终端中按 CTRL+C

（Linux 下强制结束当前进程的中断命令）组合键来关闭；如果 Elasticsearch 是后台启动，需要先获取 Elasticsearch 的进程 id，再使用结束进程的命令。查找 Elasticsearch 进程 id 可以使用 jps 命令，也可以使用 ps aux|grep elasticsearch 命令。查找到 Elasticsearch 的进程之后，使用 kill 命令终止 Elasticsearch 进程。

4.2.6　基本配置

config 目录是存放配置文件的地方，该目录下的 elasticsearch.yml 是基本配置文件，jvm.options 是虚拟机参数配置文件，log4j2.properties 是日志配置文件。Elasticsearch 的一些常用配置介绍如下：

- cluster.name: my-application
  配置 Elasticsearch 的集群名称，默认是"elasticsearch"，Elasticsearch 会自动发现在同一网段下的 Elasticsearch 节点，如果在同一网段下有多个集群，就可以用这个属性来区分不同的集群。
- node.name: node-1
  配置 Elasticsearch 的节点名，默认随机指定一个漫威漫画里的 3000 多个角色的名字。和集群名称一样可以自定义，同一个集群的集群名称要配置统一，节点名称取不同值便于区分。
- node.master: true
  指定该节点是否是 master 节点，默认是 true，Elasticsearch 默认集群中的第一台机器为 master，如果这台机出现故障就会重新选举 master。
- node.data: true
  指定该节点是否存储索引数据，默认为 true。
- index.number_of_shards: 5
  设置默认索引分片个数，默认值为 5，每个索引分成 5 个分片。在 5.0 版本以前有效，5.0 版本以后会报参数异常，提示节点中不能指定索引级别的配置。
- index.number_of_replicas: 1
  设置默认索引副本个数，默认为 1。和分片配置一样，只在 5.0 之前的版本配置生效。
- path.data: /path/to/data
  设置索引数据的存储路径，默认是 Elasticsearch 根目录下的 data 文件夹，可以设置多个存储路径，用逗号隔开，比如: path.data: /path/to/data1，/path/to/data2
- path.logs: /path/to/logs
  设置日志文件的存储路径，默认是 Elasticsearch 根目录下的 logs 文件夹
- bootstrap.mlockall: true
  设置为 true 来锁住内存。因为当 jvm 开始 swapping 时 Elasticsearch 的效率会降低，所以要保证它不 swap，可以把 ES_MIN_MEM 和 ES_MAX_MEM 两个环境变量设置成同一个值，并且保证机器有足够的内存分配给 Elasticsearch。
- network.host: 192.168.0.1
  设置绑定的 IP 地址，可以是 IPv4 或 IPv6 的，默认为 0.0.0.0。

- http.port: 9200
  设置对外服务的 HTTP 端口，默认为 9200。
- transport.tcp.port: 9300
  设置节点间交互的 TCP 端口，也是 Java API 中使用的端口，默认是 9300。
- transport.tcp.compress: true
  设置是否压缩 TCP 传输时的数据，默认为 false，不压缩。
- http.max_content_length: 100mb
  设置内容的最大容量，默认 100mb
- http.cors.enabled: false
  是否使用 HTTP 协议对外提供服务，默认为 true。
- discovery.zen.minimum_master_nodes: 1
  设置这个参数来保证集群中的节点可以知道其他 N 个有 master 资格的节点。默认为 1。
- discovery.zen.ping.timeout: 3s
  设置集群中自动发现其他节点时 ping 连接超时时间，默认为 3 秒，对于比较差的网络环境可以高该值来防止自动发现时出错。
- discovery.zen.ping.multicast.enabled: false
  设置是否打开多播发现节点，默认是 true。
- discovery.zen.ping.unicast.hosts: ["host1", "host2:port", "host3[portX-portY]"]
  设置集群中 master 节点的初始列表，可以通过这些节点来自动发现新加入集群的节点。
- script.engine.groovy.inline.update: on
  开启 groovy 脚本支持，inline 表示脚本的来源为内嵌式，还可以设为 stored 或者 file。update 表示允许脚本语言执行更新操作，也可以设置其他操作，比如 search、eggs 等。
- script.inline: true
  简写方式，开始所有脚本语言行内执行所有支持的操作。

4.3　REST 命令

Elasticsearch 提供用于各种任务的 RESTful API，我们首先对 REST 架构做一个全面的了解。REST 全称是 Representational State Transfer，翻译为表述性状态转移，源自 Roy Thomas Fielding 博士 2000 年在加州大学欧文分校就读期间发表的著名博士论文《架构风格与基于网络应用软件的架构设计》。迄今为止，关于 REST 最系统最全面的论述仍是 Fielding 博士的论文。

REST 架构风格最重要的架构约束有以下 6 个：

（1）采用客户-服务器（Client-Server）架构，通信只能由客户端单方面发起，表现为请求-响应的形式。

（2）通信的会话状态由客户端负责维护。

（3）响应内容可以在通信链的某处被缓存，以改善网络效率。

（4）通信链的组件之间通过统一的接口相互通信，以提高交互的可见性。

（5）采用分层系统（Layered System）通过限制组件的行为（即，每个组件只能“看到”与其交互的紧邻层），将架构分解为若干等级的层。

（6）支持通过下载并执行一些代码（例如 Java Applet、Flash 或 JavaScript），对客户端的功能进行扩展。

REST 有以下优点：

- 可更高效地利用缓存来提高响应速度。
- 通信本身的无状态性可以让不同的服务器处理一系列请求中的不同请求，提高服务器的扩展性。
- 浏览器可作为客户端，简化软件需求。
- 相对于其他叠加在 HTTP 协议之上的机制，REST 的软件依赖性更小。
- 不需要额外的资源发现机制。
- 兼容性更好。

符合 REST 设计风格的 Web API 称为 RESTful API，REST 是设计风格而不是标准。REST 通常基于使用 HTTP、URI 和 XML 以及 HTML 这些现有的广泛流行的协议和标准。在一个类 REST 的架构中资源是由 URI 来指定的，对资源的操作包括获取、创建、修改和删除，这些操作正好对应 HTTP 协议提供的 GET、POST、PUT 和 DELETE 方法。资源的表现形式可以是 XML、HTML、JSON 或其他的任何格式，通过操作资源的表现形式来操作资源。表 4-3 列举了 HTTP 请求方法在 RESTful API 中的典型应用。

表4-3　HTTP请求方法在RESTful API中的典型应用

方法 \ 资源	一组资源的 URI，比如 http://example.com/resources	单个资源的 URI，比如 http://example.com/resources/1
GET	列出 URI，以及该资源组中每个资源的详细信息（后者可选）	获取指定资源的详细信息，格式可以自选一个合适的网络媒体类型（比如，XML、JSON 等）
PUT	使用给定的一组资源替换当前整组资源	替换/创建指定的资源，并将其追加到相应的资源组中
POST	在本组资源中创建/追加一个新的资源。该操作往往返回新资源的 URL	把指定的资源当作一个资源组，并在其下创建/追加一个新的元素，使其隶属于当前资源
DELETE	删除整组资源	删除指定的元素

4.3.1　CURL 工具

CURL 是利用 URL 语法在命令行方式下工作的开源文件传输工具，被广泛应用在 Unix、多种 Linux 发行版中，并且有 DOS 和 Win32、Win64 下的移植版本，支持 FTP、FTPS、HTTP、HTTPS、IMAP 、POP3 等十几种通信协议。下面简介一下在 Windows、Ubuntu、Mac OS X 系统上安装 CURL 工具的方法：

- Windows 系统安装 CURL

如果使用的是 Windows 操作系统，首先到 CURL 官网（https://curl.haxx.se）的下载页面找到与当前所使用的系统对应的安装包，解压后完成安装。

- Ubuntu 系统安装 CURL

如果使用的是 Ubuntu 系统，打开终端，执行如下安装命令:

```
sudo apt-get install curl libcurl3 libcurl3-dev php5-curl.
```

命令执行完毕后需重启系统。

- Mac OS X 系统安装 CURL

Mac OS X 系统自带 CURL 工具，可以在终端执行命令:curl -V，查看 CURL 工具的版本信息。

使用 CURL 发送 GET 请求查看一条文档，执行命令和返回结果如图 4-15 所示。

```
1. bash
Bee:~ bee$ curl -XGET "http://172.31.44.9:9200/blog/article/1?pretty"
{
  "_index" : "blog",
  "_type" : "article",
  "_id" : "1",
  "_version" : 1,
  "found" : true,
  "_source" : {
    "title" : "master jvm",
    "postdate" : "2017-08-01"
  }
}
Bee:~ bee$
```

图 4-15　CURL 命令查看文档

4.3.2　Kibana Dev Tools

Kibana 是 Elastic 公司推出的一个针对 Elasticsearch 的开源分析及可视化平台，可以搜索、查看存放在 Elasticsearch 索引里的数据，还可以绘制多种图表用于高级数据分析与可视化。Kibana Dev Tools 是 Kibana 提供的一个开发者工具，可以方便地执行 REST 请求，具有自动提示和自动补全功能。Kibana Dev Tools 的前身是 Chrome 的 Sense 插件，唯一的缺点是对中文的支持不是太好。Kibana 的安装步骤如下：

步骤 01 下载 Kibana 安装包：https://www.elastic.co/downloads/kibana

步骤 02 解压安装包：tar –zxvf kibana-5.4.0-darwin-x86_64.tar.gz

步骤 03 修改配置文件。修改 Kibana 根目录下 config 文件夹中的 kibana.yml，配置 elasticsearch.url 的值为 Elasticsearch 的访问 IP。

步骤 04 切换到 kibana 根目录，执行启动命令

```
./bin/kibana
```

步骤 05 访问 5601 端口：http://localhost:5601

Kibana 启动成功后，首页如图 4-16 所示，“Configure an index pattern”是配置要进行数据分析的索引名规则，支持正则表达式。单击“Dev Tools”菜单栏，界面如图 4-17 所示，左侧输入 REST 命令，单击绿色箭头即可执行操作，右侧面板显示操作的结果。

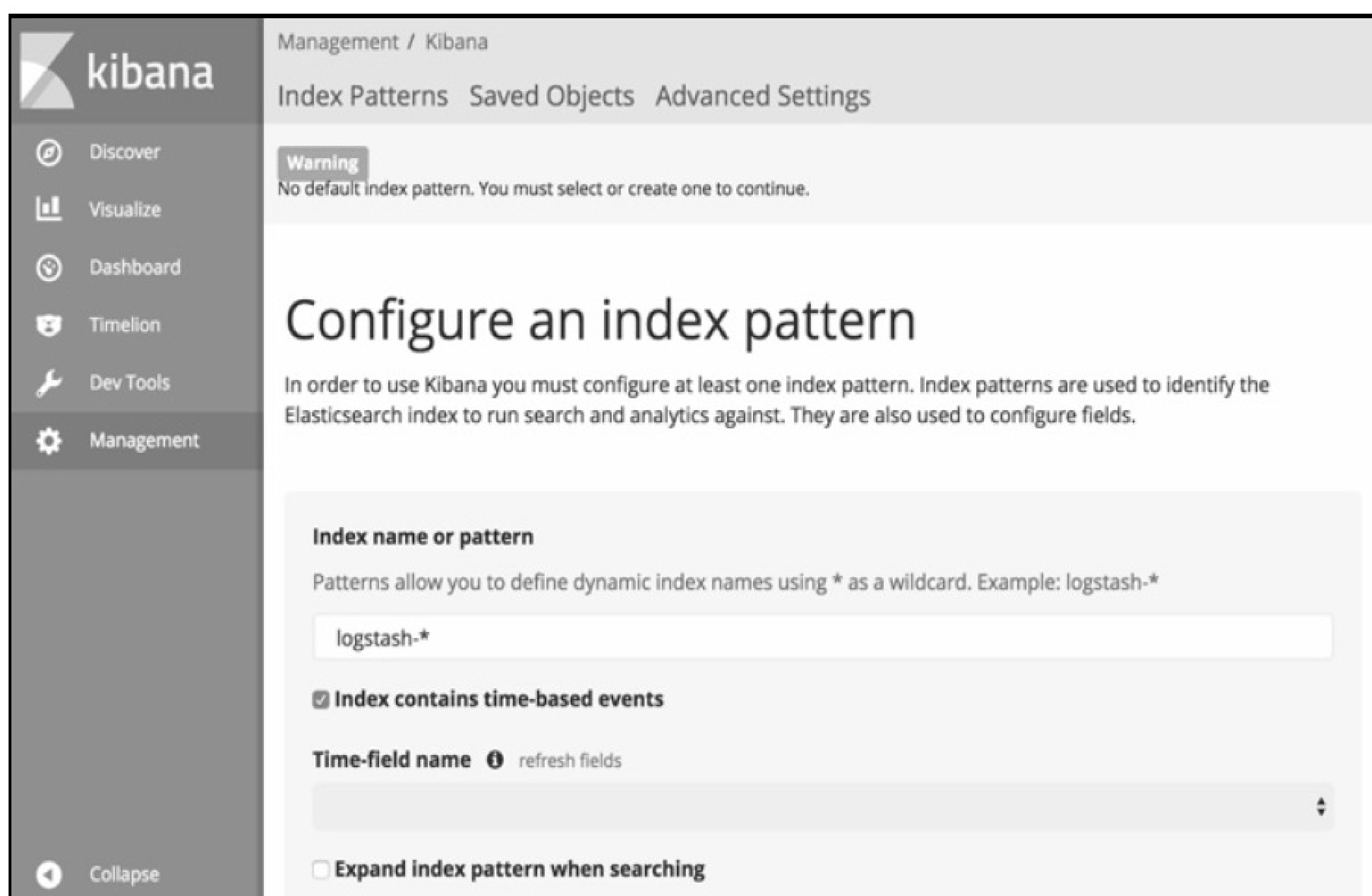


图 4-16　Kibana 启动首页

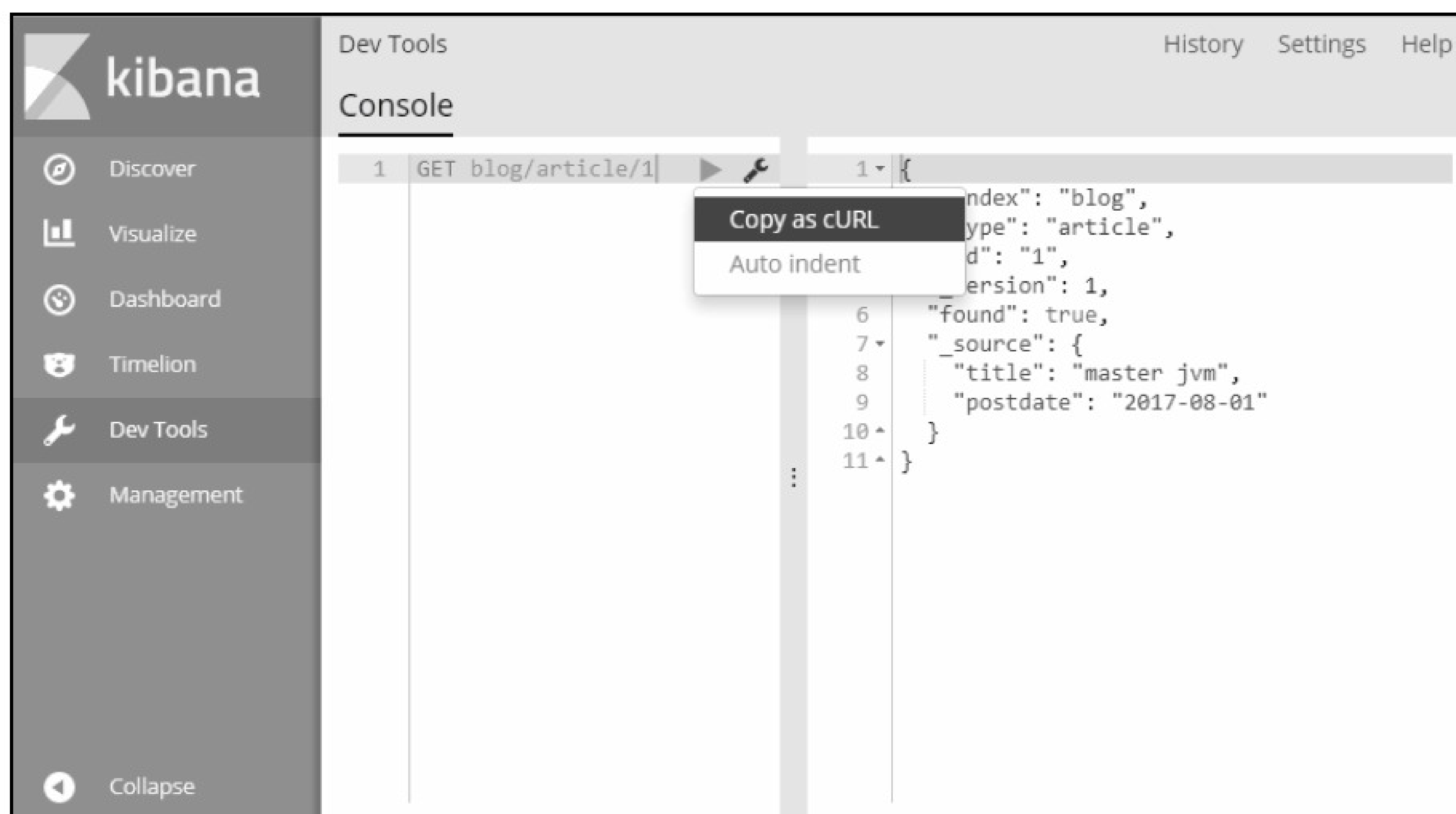


图 4-17　Kibana 开发者工具

4.4　中文分词器配置

4.4.1　IK 分词器安装

Elasticsearch 作为开源搜索引擎服务器，其核心功能在于搜索数据。索引是把文档写入 Elasticsearch 的过程，搜索是匹配查询条件找出文档的过程，实现全文检索一个分析过程，分析过程主要分为两步，第一步是词条化，分词器把输入文本转化为一个个的词条流；第二步是

过滤，在这个阶段有若干个过滤器处理词条流中的词条，比如停用词过滤器会从词条流中去除不相干的词条，同义词过滤器会添加新词条或者改变已有词条，小写过滤器会把所有词条变成小写。

Elasticsearch 内置多种分词器可供使用，在索引和查询过程中我们可以指定分词器，也可以通过安装插件的方式使用第三方分词工具。Elasticsearch 内置的分词器简介如下：

- Standard Analyzer：标准分词器会把句子分成一个个的单词，对于大多数欧洲语言（比如英语）进行分词非常合适。
- Simple Analyzer：简单分词器基于非字母字符进行分词，单词会被转化为小写字母。
- Whitespace Analyzer：空格分词器遇到空格就进行切分。
- Stop Analyzer：与简单分词器类似，增加了停用词过滤功能。
- Keyword Analyzer：关键词分词器非常简单，输入文本和输出文本全部相同。
- Pattern Analyzer：利用正则表达式对文本进行灵活划分，单词会被小写，支持停用词。
- Language Analyzers：对特定语言的分词器，比如英语、法语。
- Fingerprint Analyzer：指纹分析仪分词器通过创建标记进行重复检测。

Elasticsearch 中文分词器业界使用最多的是 elasticsearch-analysis-ik，它是 Elasticsearch 的一个第三方插件，代码托管在 GitHub 上，项目地址为：https://github.com/medcl/elasticsearch-analysis-ik，统一版本号之后 IK 的版本也要与 Elasticsearch 的版本一致。

IK 分词器的安装步骤如下：

步骤 01 打开网址 https://github.com/medcl/elasticsearch-analysis-ik/releases，选择 5.4.0 版本，找到 elasticsearch-analysis-ik-5.4.0.zip，下载到本地并解压缩。

步骤 02 在 elasticsearch-5.4.0/plugins/目录下新建名为 ik 的文件夹，拷贝 elasticsearch- analysis-ik-5.4.0 目录下的所有文件到 elasticsearch-5.4.0/plugins/ik/目录下。

最后，重启 Elasticsearch 服务，启动过程没有报错或异常，日志输出信息中有如下提示说明 IK 分词器插件安装成功：

```
[INFO ][ik-analyzer    ] [Dict Loading] ik/custom/mydict.dic
[INFO ][ik-analyzer    ] [Dict Loading] ik/custom/single_word_low_freq.dic
[INFO ][ik-analyzer    ] [Dict Loading] ik/custom/ext_stopword.dic
```

如果想通过编译源码的方式安装，步骤如下：

步骤 01 下载 IK 源码文件，以编译 master 分支上最新的版本为例，把源码下载到本地，命令如下：git clone https://github.com/medcl/elasticsearch-analysis-ik.git

步骤 02 切换到根目录，运行 mvn 打包命令 mvn package，打包完成以后根目录下会生成一个 target 文件夹。

步骤 03 target/releases 目录下的压缩包即为 IK 的安装文件。

4.4.2 扩展本地词库

网络流行语层出不穷，比如“洪荒之力”“蓝瘦香菇”等。在没有加入自定义词库之前，测试分词效果：

```
PUT test
GET /test/_analyze?analyzer=ik_smart
{
  "text": "洪荒之力"
}
```

分词结果：

```
{
  "tokens": [
    {
      "token": "洪荒",
      "start_offset": 0,
      "end_offset": 2,
      "type": "CN_WORD",
      "position": 0
    },
    {
      "token": "之力",
      "start_offset": 2,
      "end_offset": 4,
      "type": "CN_WORD",
      "position": 1
    }
  ]
}
```

在 elasticsearch-5.4.0/plugins/ik/config/ik/custom 目录下，新增文件 hotwords.dic，在文件中添加词语“洪荒之力”，每一个词语一行，最后在 IK 插件的配置文件（elasticsearch-5.4.0/plugins/ik/config/IKAnalyzer.cfg.xml）中指定新增的词库位置。要想使自定义词库生效，需要重启 Elasticsearch。如果配置成功，在 Elasticsearch 启动的输出日志中可以看到加载信息:

```
[INFO ][ik-analyzer  ] [Dict Loading] ik/custom/hotwords.dic
```

再次运行分词测试命令，分词结果如下：

```
{
  "tokens" : [ {
    "token" : "洪荒之力",
    "start_offset" : 0,
    "end_offset" : 4,
    "type" : "CN_WORD",
    "position" : 0
  } ]
}
```

扩展本地停用词词库的方法与之类似，新增停用词字典，在配置文件中指定停用词字典位置，重启 Elasticsearch 即可。

4.4.3　配置远程词库

新建一个 Java Web 工程，把词库放在工程 WebContent 目录下，部署到 Tomcat 下，确保能正常访问，如图 4-18 所示。

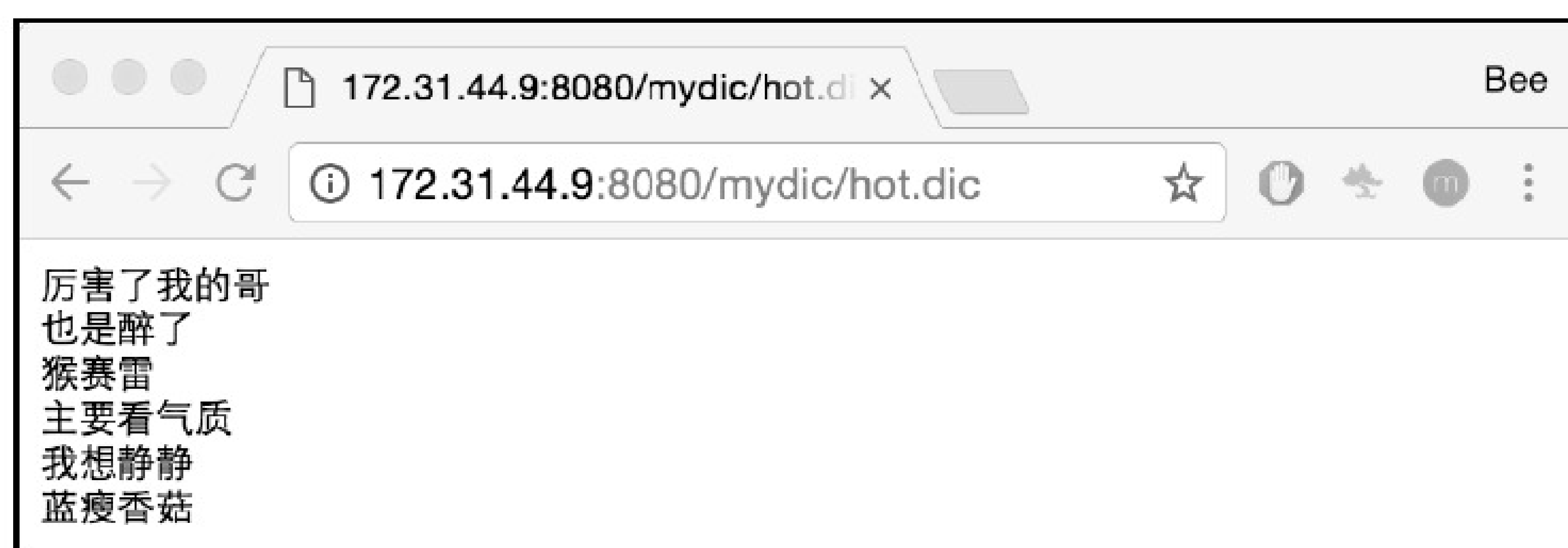


图 4-18　远程访问自定义词库

在字典文件中按行写入新词，同样在 ik 插件的配置文件中指定新增的远程词库地址，然后重新启动 Elasticsearch。当字典文件中有新词增加时，IK 分词插件会自动重新加载词典，在日志中会输出加载信息，如图 4-19 所示。

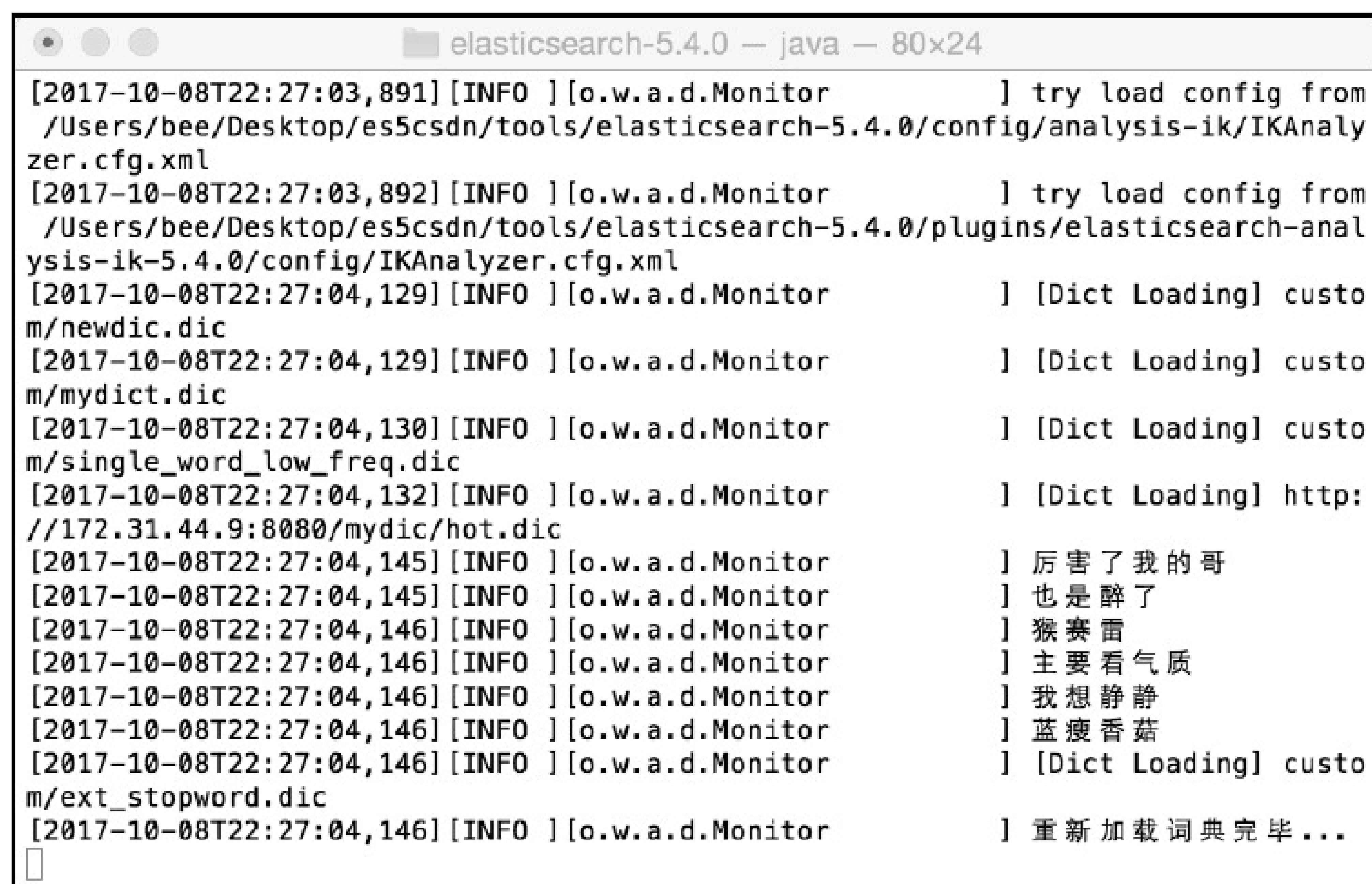


图 4-19　日志中输出远程词库信息

测试分词效果，执行命令：

```
GET /test/_analyze?analyzer=ik_smart
{
    "text":"蓝瘦香菇"
}
```

返回结果如下：

```
{
  "tokens" : [ {
    "token" : "蓝瘦香菇",
    "start_offset" : 0,
    "end_offset" : 4,
    "type" : "CN_WORD",
    "position" : 0
  } ]
}
```

4.5　Head 插件使用指南

Elasticsearch-Head（简称 Head）是一个 HTML 5 编写的集群操作和管理工具，可以对集群进行傻瓜式操作。在 Head 插件中可以显示集群的拓扑结构，执行索引和节点级别的操作，同时也能够输入 RESTful 命令和 Elasticsearch 交互。5.0 之前的版本可以通过 Elasticsearch 的 plugin 命令安装，5.0 版本之后的 Head 插件独立运行。

4.5.1　Head 插件的安装

安装步骤如下：

步骤 01　配置 Node 环境。

由于 Head 插件是采用 HTML 5 编写的，它的运行需要 Node.js 环境，到 Node.js 官网 http://nodejs.org 下载安装即可。npm 是 Node.js 的包管理工具，通过 npm 可以下载 Node.js 的模块包，npm 在安装 Node.js 时顺带已经安装成功了。此外，还需要安装 Grunt，它是一个基于命令的 Javascript 工程命令行构建工具，使用 npm 安装 Grunt 的安装命令如下：

```
npm install -g  grunt-cli
```

如图 4-20 所示，分别使用 node -v、npm -v、grunt -version 命令查看各自的版本信息，如正确返回版本号，说明 Node 环境配置成功。

```
2. bash
Bee:elasticsearch-head-master bee$ node -v
v6.9.4
Bee:elasticsearch-head-master bee$ nmp -v
-bash: nmp: command not found
Bee:elasticsearch-head-master bee$ grunt -version
grunt-cli v1.2.0
grunt v1.0.1
Bee:elasticsearch-head-master bee$ sudo npm install -g cnpm --registry=https://
registry.npm.taobao.org
/usr/local/bin/cnpm -> /usr/local/lib/node_modules/cnpm/bin/cnpm
/usr/local/lib
└── cnpm@5.1.1

Bee:elasticsearch-head-master bee$
```

图 4-20　Node 环境配置

步骤 02 下载 Head 插件源码。

Head 插件项目地址为 https://github.com/mobz/elasticsearch-head，下载源码到本地，切换到 elasticsearch-head-master 目录下，运行命令：

```
npm install
```

如果速度较慢或者安装失败，可以使用国内镜像：

```
sudo npm install -g cnpm --registry=https://registry.npm.taobao.org
```

步骤 03 修改 Elasticsearch 配置文件。

编辑 elasticsearch-5.4.4/config/elasticsearch.yml，加入以下内容：

```
http.cors.enabled: true
http.cors.allow-origin: "*"
```

作用是开启 HTTP 对外提供服务，使 Head 插件能够访问 Elasticsearch 集群，修改完成之后需重启 Elasticsearch。

步骤 04 修改 Head 插件配置文件。

打开 elasticsearch-head-master/Gruntfile.js，找到下面 connect 属性，修改 hostname 的值为 Elasticsearch 的访问 IP：

```
connect: {
            server: {
                options: {
                    hostname: 'localhost',
                    port: 9100,
                    base: '.',
                    keepalive: true
                }
            }
        }
```

步骤 05 启动 Head 插件。

切换到 elasticsearch-head-master/目录下，运行启动命令：

```
grunt server
```

启动成功之后的输出信息如图 4-21 所示。

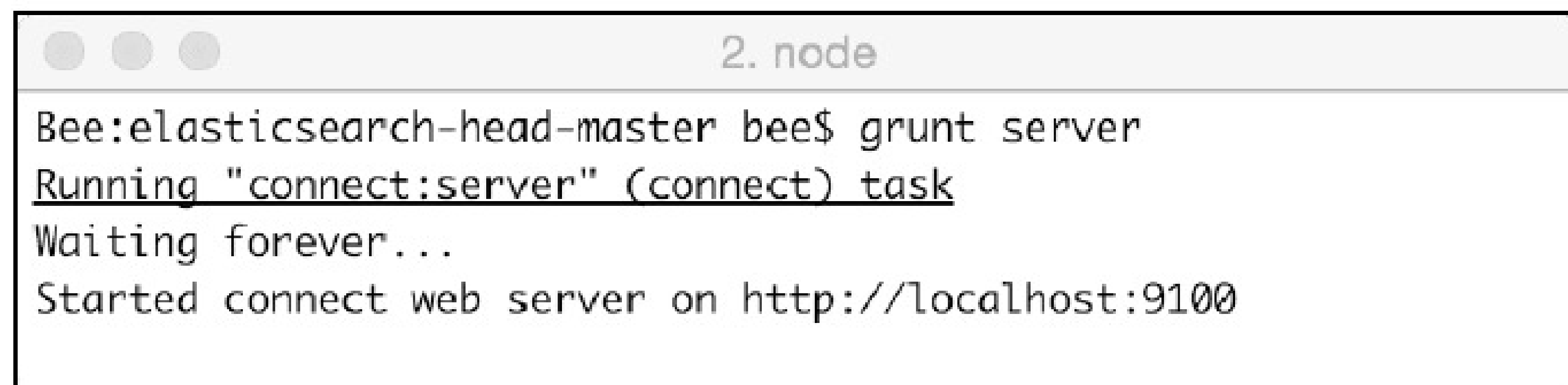


图 4-21　运行 Head 插件

访问 9100 端口即可看到如图 4-22 所示的界面（spnews 和 blog 是笔者创建的两个索引）。



图 4-22　Head 插件首页

4.5.2　Head 插件的使用

1. 概览界面

访问 Head 插件首先看到的是概览选项卡，在概览界面可以查看当前集群中的节点个数、索引信息、集群健康信息等。如图 4-22 所示，当前集群名称为 Elasticsearch，该集群有 1 个节点，节点名称为 CHo4z-e，有星星标记的为 master 节点。spnews 和 blog 是 Elasticsearch 中存在的两个索引，索引的大小、文档个数都在索引名下面展示出来了，单击信息和动作按钮还可以执行相应的操作。黄色背景显示的集群健康值为 yellow，创建的 blog 索引默认副本数为 1，现在只启动了一个节点，副本数据没有分配，因此集群健康值是黄色的。“8 of 13”表示集群中共有 13 个分片，已分配 8 个，还有 5 个是未分配状态，图中绿色的方块代表可用的索引分片。

2. 索引查看界面

在索引界面可以查看集群中索引的大小和文档个数，如图 4-23 所示。



图 4-23　Head 插件索引选项卡

如图 4-24 所示，单击新建索引按钮可以创建新的索引，设置好索引名称、分片数、副本数，单击 OK 按钮以后就可以创建一个索引。



图 4-24　Head 插件中创建索引

3. 数据浏览界面

如图 4-25 所示，切换到数据浏览界面可以查看 Elasticsearch 中的索引、类型、文档的详细信息，单击左侧面板还可以根据索引、类型进行筛选。

图 4-25　Head 插件数据浏览界面

4. 基本查询选项卡

基本查询界面如图 4-26 所示，可以进行布尔查询运算。图中的操作是查询 spnews 索引中的文档，查询条件是 title 字段中一定包含关键词“足球”，单击“+”还可以添加多个查询条件，单击 search 按钮进行搜索，搜索结果会在底部显示出来。勾选“显示查询语句”复选框可以看到一个 JSON 格式的查询字符串，这也是 Elasticsearch 接收并查询的搜索语句。

图 4-26　Head 基本查询界面

5. 复合查询

复合查询界面如图 4-27 所示，在左侧查询面板中输入 Elasticsearch 服务器地址、要查询的索引和要执行的操作（_search 表示搜索操作），空白部分写入查询语句，勾选“易读”按钮可以格式化 JSON 字符串，单击“验证 JSON”可以对查询语句的格式进行验证，单击“提交请求”即可执行查询，右侧面板中返回查询结果。如果需要保留多个复合查询，可以单击复合查询菜单旁的“+”。



图 4-27　复杂查询

4.6　本章小结

本章对 Elasticsearch 进行了概述，介绍了 Lucene 和 Elasticsearch 的关系、Elasticsearch 的核心概念和安装配置步骤、中文分词器的安装与配置、Head 和 Kibana 的使用方法以及如何向 Elasticsearch 发送 REST 命令。

第 5 章

Elasticsearch 集群入门

本章学习要点：

* Elasticsearch 索引管理
* Elasticsearch 文档管理
* Elasticsearch 映射详解

5.1 索引管理

本节从新建一个索引开始 Elasticsearch 的学习之旅，包括如何进行索引的创建、索引的删除、副本的更新、索引读写权限以及索引别名的配置。

5.1.1 新建索引

创建索引和在 MySQL 中创建一个数据库是一样的，注意 Elasticsearch 索引名称中不能出现大写字母。例如新建一个索引，名为 blog，命令如下：

```
PUT blog
```

响应结果：

```
{
  "acknowledged": true,
  "shards_acknowledged": true
}
```

返回结果显示 acknowledged 的值为 true，说明新建索引成功。如果索引名含有大写字母，会报一个非法的索引名异常，测试命令如下：

```
PUT Abc
```

响应结果：

```
{
  "error": {
    "root_cause": [
      {
        "type": "invalid_index_name_exception",
        "reason": "Invalid index name [Abc], must be lowercase",
        "index_uuid": "_na_",
        "index": "Abc"
      }
    ],
    "type": "invalid_index_name_exception",
    "reason": "Invalid index name [Abc], must be lowercase",
    "index_uuid": "_na_",
    "index": "Abc"
  },
  "status": 400
}
```

如果新添加的索引在 Elasticsearch 服务器中已存在，新建同名索引会报索引已存在异常，再一次执行创建名为 blog 的索引命令：

```
PUT blog
```

响应结果：

```
{
  "error": {
    "root_cause": [
      {
        "type": "index_already_exists_exception",
        "reason": "index [blog/-Mf4cVKWRm6TJIFjieeWhw] already
                 exists",
        "index_uuid": "-Mf4cVKWRm6TJIFjieeWhw",
        "index": "blog"
      }
    ],
    "type": "index_already_exists_exception",
    "reason": "index [blog/-Mf4cVKWRm6TJIFjieeWhw] already exists",
    "index_uuid": "-Mf4cVKWRm6TJIFjieeWhw",
    "index": "blog"
  },
  "status": 400
}
```

Elasticsearch 默认给一个索引设置 5 个分片 1 个副本，一个索引的分片数一经指定后就不能再修改，副本数可以通过命令随时修改。如果想创建自定义分片数和副本数的索引，可以通过 setting 参数在索引时设置初始化信息。以创建 3 个分片 0 个副本名为 blog 的索引为例，命令如下：

```
PUT blog
{
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 0
  }
}
```

5.1.2　更新副本

Elasticsearch 支持修改一个已存在索引的副本数，把 blog 索引的副本数设置为 2，命令如下：

```
PUT blog/_settings
{
   "number_of_replicas": 2
}
```

5.1.3　读写权限

除了设置分片和副本，还可以对索引的读写操作进行限制，下面是三个读写权限的参数：

- blocks.read_only:true　设置当前索引只允许读不允许写或者更新。
- blocks.read:true　禁止对当前索引进行读操作。
- blocks.write:true　禁止对当前索引进行写操作。

以禁止 blog 索引进行写操作为例，执行命令如下：

```
PUT blog/_settings
{
  "blocks.write": true
}
```

命令执行完成以后就不能再往 blog 索引中写入数据，否则会报索引锁定异常，测试写入一条文档：

```
PUT blog/article/1
{
  "title": " Java 虚拟机"
}
```

返回结果：

```
{
  "error": {
```

```
    "root_cause": [
      {
        "type": "cluster_block_exception",
        "reason": "blocked by: [FORBIDDEN/8/index write (api)];"
      }
    ],
    "type": "cluster_block_exception",
    "reason": "blocked by: [FORBIDDEN/8/index write (api)];"
  },
  "status": 403
}
```

恢复索引的写入权限，只需把索引的 blocks.write 属性设置为 false 即可，命令如下：

```
PUT blog/_settings
{
  "blocks.write": false
}
```

5.1.4 查看索引

使用 GET 方法加上_setting 参数可以查看一个索引的所有配置信息，例如查看 blog 索引所有的设置信息，命令如下：

```
GET blog/_settings
```

返回结果：

```
{
  "blog": {
    "settings": {
      "index": {
        "number_of_shards": "3",
        "blocks": {
          "write": "false"
        },
        "provided_name": "blog",
        "creation_date": "1503133584844",
        "number_of_replicas": "2",
        "uuid": "0yQX5PqNTtKmwRQouFsd8Q",
        "version": {
          "created": "5040099"
        }
      }
    }
  }
}
```

同时查看多个索引的 setting 信息，命令如下：

```
GET blog,twitter/_settings
```

查看集群中所有索引的 setting 信息，命令如下：

```
GET _all/_settings
```

5.1.5　删除索引

索引的删除只需要使用 DELETE 方法，传入要删除的索引名即可，执行索引删除命令之前需要慎重考虑，因为一旦执行删除操作索引中的文档就不复存在，注意在删除重要数据之前做好备份工作。

删除名为 blog 的索引，命令如下：

```
DELETE blog
```

如果删除成功，会有以下响应：

```
{
  "acknowledged": true
}
```

如果删除的索引名不存在，会报索引未找到异常。尝试删除一个不存在的索引，执行命令：

```
DELETE  bloga
```

响应结果：

```
{
  "error": {
    "root_cause": [
      {
        "type": "index_not_found_exception",
        "reason": "no such index",
        "resource.type": "index_or_alias",
        "resource.id": "bloga",
        "index_uuid": "_na_",
        "index": "bloga"
      }
    ],
    "type": "index_not_found_exception",
    "reason": "no such index",
    "resource.type": "index_or_alias",
    "resource.id": "bloga",
    "index_uuid": "_na_",
    "index": "bloga"
```

```
  },
  "status": 404
}
```

5.1.6　索引的打开与关闭

Elasticsearch 中的索引可以进行打开和关闭操作，一个关闭了的索引几乎不占用系统资源。关闭一个索引的命令如下：

```
POST blog/_close
```

索引关闭后，Head 插件中会显示关闭状态的索引，已关闭的索引不能进行读写操作。一个关闭的索引可以重新开启，命令如下：

```
POST blog/_open
```

同时关闭或开启多个索引也是允许的，以同时关闭 testa、testb、testc 三个索引为例，命令如下：

```
POST testa,testb,testc/_close
```

同时打开三个索引：

```
POST testa,testb,testc/_open
```

如果 Elasticsearch 集群中不存在开启/关闭请求中的全部索引，将会抛出索引不存在错误，此时可以通过 ignore_unavailable 参数来操作只存在的索引，命令如下：

```
POST testa,testb,testc/_close?ignore_unavailable=true
```

索引的开关操作也支持通配符和_all，关闭集群中所有索引的命令如下：

```
POST _all/_close
```

关闭以 test 开头的索引：

```
POST test*/_close
```

5.1.7　复制索引

_reindex API 可以把文档从一个索引（源索引）复制到另一个索引（目标索引），目标索引不会复制源索引中的配置信息，_reindex 操作之前需要设置目标索引的分片数、副本数等信息。

把 blog 索引的文档复制到 blog_new 索引中的命令如下：

```
POST _reindex
{
  "source": { "index": "blog"},
  "dest": {"index": "blog_news"}
}
```

上述命令会把源索引中的所有文档都复制到目标索引中，也可以在源索引中增加 type 和

query 来限制文档，下面把 blog 索引 article 类型下 title 中含有 git 关键字的文档复制的 blog_new 索引中，命令如下：

```
POST _reindex
{
  "source": {
    "index": "blog",
    "type":"article",
    "query":{
      "term": { "title": "git"}
    }
  },
  "dest": {
    "index": "blog_news"
  }
}
```

5.1.8　收缩索引

一个索引的分片初始化以后是无法再做修改的，但可以使用 shrink index AP 提供的缩小索引分片数机制，把一个索引变成一个更少分片的索引，但是收缩后的分片数必须是原始分片数的因子，比如有 8 个分片的索引可以收缩为 4、2、1，有 15 个分片的索引可以收缩为 5、3、1，如果分片数为素数（7、11 等），那么只能收缩为 1 个分片。收缩索引之前，索引中的每个分片都要在同一个节点上。收缩索引的完成过程如下：

首先，创建一个新的目标索引，设置与源索引相同，但新索引的分片数量较少。

然后，把源索引硬链接到目标索引。（如果文件系统不支持硬链接，那么所有段都被复制到新索引中，这是一个耗费更多时间的过程。）

最后，新的索引恢复使用。

在缩小索引之前，索引必须被标记为只读，所有分片都会复制到一个相同的节点并且节点健康值为绿色的。这两个条件可以通过下列请求实现，以收缩 blog 索引为例，命令如下：

```
PUT blog/_settings
{
  "index.routing.allocation.require._name": "shrink_node_name",
  "index.blocks.write": true
}
```

blog_new 为目标索引，在收缩时可指定目标索引的分片数、副本数等配置信息，命令如下：

```
POST blog/_shrink/blog_new
{
  "settings": {
    "index.number_of_replicas": 0,
    "index.number_of_shards": 1,
```

```
    "index.codec": "best_compression"
  },
  "aliases": {
    "my_search_indices": {}
  }
}
```

5.1.9　索引别名

索引别名就是给一个索引或者多个索引起的另一个名字。为名为 test1 的索引创建别名 alias1，命令格式如下：

```
POST /_aliases
{
    "actions" : [
        { "add" : { "index" : "test1", "alias" : "alias1" } }
    ]
}
```

移除别名：

```
POST /_aliases
{
    "actions" : [
        { "remove" : { "index" : "test1", "alias" : "alias1" } }
    ]
}
```

一次给多个索引创建同一个别名：

```
POST /_aliases
{
    "actions" : [
        { "add" : { "index" : "test1", "alias" : "alias1" } },
        { "add" : { "index" : "test2", "alias" : "alias1" } }
    ]
}
```

上述命令也可以使用简写形式，命令如下：

```
POST /_aliases
{
    "actions" : [
        { "add" : { "indices" : ["test1", "test2"], "alias" : "alias1" } }
    ]
}
```

同样也可以一次性移除别名：

```
POST /_aliases
{
   "actions" : [
   { "remove" : { "indices" : ["test1", "test2"], "alias" :    "alias1" } }
   ]
}
```

增加别名和移除别名也可以混合使用：

```
POST /_aliases
{
   "actions" : [
       { "remove" : { "index" : "test1", "alias" : "alias1" } },
       { "add" : { "index" : "test2", "alias" : "alias1" } }
   ]
}
```

如图 5-1 所示，给索引 spnews 和 blog 创建别名 myblog，在 head 插件中会有索引别名的显示，设置别名以后，对别名 myblog 进行操作等价于对索引 spnews 和 blog 进行操作。



图 5-1　Head 插件中查看索引别名

需要注意的是，如果别名和索引是一对一的，使用别名索引文档或者根据 ID 查询文档是可以的，但是如果别名和索引是一对多的，使用别名会发生错误，因为 Elasticsearch 不知道把文档写入哪个索引中去或者从哪个索引中读取文档。执行查看文档命令：

```
GET myblog/article/1
```

返回结果：

```
{
  "error": {
    "root_cause": [
      {
        "type": "illegal_argument_exception",
        "reason": "Alias [myblog] has more than one indices associated with it
                 [[blog, spnews]], can't execute a single index op"
```

```
        }
      ],
      "type": "illegal_argument_exception",
      "reason": "Alias [myblog] has more than one indices associated
          with it [[blog, spnews]], can't execute a single index op"
    },
    "status": 400
  }
```

Elasticsearch 支持通过通配符同时给多个索引设置别名，假设集群中存在 test、testa、testb 三个索引，执行命令给所有以 test 开头的索引设置别名，命令如下：

```
POST /_aliases
{
    "actions" : [{ "add" : { "index" : "test*", "alias" : "mytest" } }]
}
```

在 Head 插件中查看效果，如图 5-2 所示，可以看到 test、testa、testb 三个索引有共同的别名 mytest。

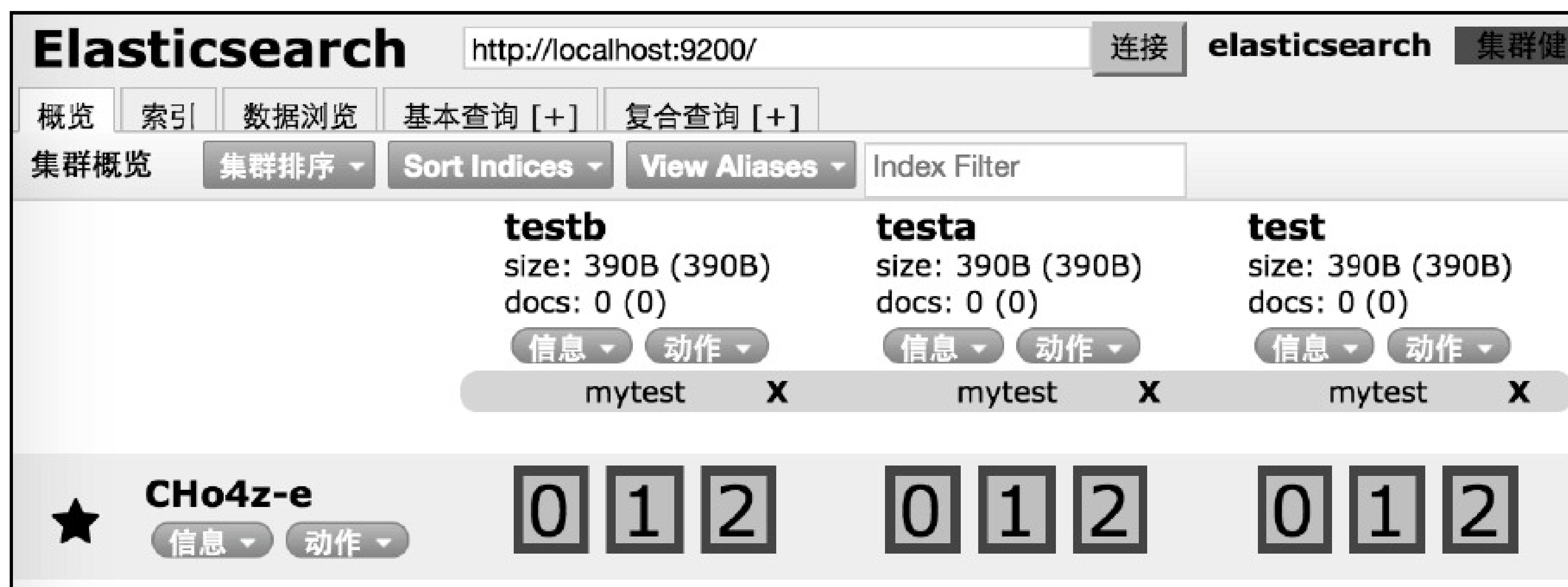


图 5-2　Head 插件中查看索引共有别名

如果想要查看某一个索引的别名是什么，比如查看上例中索引 test 的别名，可执行以下命令：

```
GET /test/_aliases
```

返回结果：

```
{
  "test": {
    "aliases": { "mytest": {}}
  }
}
```

查看“mytest”对应哪些索引的别名，执行命令：

```
GET /mytest/_aliases
```

返回结果：

```
{
  "test": {
    "aliases": {"mytest": {}}
  },
  "testb": {
      "aliases": {"mytest": {}}
  },
  "testa": {
    "aliases": {"mytest": {}}
  }
}
```

查看 Elasticsearch 集群上所有的可用别名，执行命令：

```
GET /_aliases
```

返回结果：

```
{
  "test": {
    "aliases": { "mytest": {}}
  },
  "spnews": {
    "aliases": {"myblog": {}}
  },
  "blog": {
    "aliases": {"myblog": {}}
  },
  ".kibana": {
    "aliases": {}
  }
}
```

5.2 文档管理

Elasticsearch 中文档的增删改查和关系型数据库操作非常相似，当然 Elasticsearch 作为一个搜索引擎服务器功能远不仅如此。

5.2.1 新建文档

准备一条 JSON 格式博客文章，包括文章的 id、标题、发布时间、文章内容。

```
PUT blog/article/1
{
    "id": 1,
    "title": "Git 簡介",
    "posttime": "2017-05-01",
    "content": "Git 是一款免费、开源的分布式版本控制系统"
}
```

如果没有出现错误，Elasticsearch 服务器会返回一个 JSON 格式的响应。

```
{
  "_index": "blog",
  "_type": "article",
  "_id": "1",
  "_version": 1,
  "result": "created",
  "_shards": {
    "total": 1,
    "successful": 1,
    "failed": 0
  },
  "created": true
}
```

响应信息中包含创建的文档所在的索引（index）、类型（type）、id、版本、分片、是否创建成功等信息。其中 index/type/id 用于确定文档所在的位置，版本号会随着文档的更新自动递增，shards 用于显示分片信息。

如果不指定文档的 id，Elasticsearch 会自动生成它，把上面的命令去掉 id 参数，然后再次执行索引文档命令：

```
POST blog/article
{
    "id": 1,
    "title": "Git 簡介",
    "posttime": "2017-05-01",
    "content": "Git 是一款免费、开源的分布式版本控制系统"
}
```

Elasticsearch 响应结果：

```
{
  "_index": "blog",
  "_type": "article",
  "_id": "AV37Fs3T7NCuL2SahBa1",
  "_version": 1,
  "result": "created",
```

```
  "_shards": {
    "total": 1,
    "successful": 1,
    "failed": 0
  },
  "created": true
}
```

从响应结果中可以看到文档创建成功了，id 是自动生成的字符串。采用这种方式创建文档要使用 POST 方法，使用 PUT 方法会出现异常。

5.2.2　获取文档

Elasticsearch 提供了 GET API 查看存储在 Elasticsearch 服务器中的文档，使用 GET 命令并指定文档所在的索引、类型和 id 即可返回一个 JSON 格式的文档，命令如下：

```
GET blog/article/1
```

响应结果：

```
{
  "_index": "blog",
  "_type": "article",
  "_id": "1",
  "_version": 1,
  "found": true,
  "_source": {
    "id": 1,
    "title": "Git 简介",
    "posttime": "2017-05-01",
    "content": "Git 是一款免费、开源的分布式版本控制系统"
  }
}
```

返回结果中前 4 个属性表明文档的位置和版本信息，found 属性用于表明是否查询到文档，_source 字段中是文档的内容。

同样，如果要看一下文档不存在的情况，可执行查询命令，查询一个不存在的 id：

```
GET blog/article/100
```

响应信息：

```
{
  "_index" : "blog",
  "_type" : "article",
  "_id" : "100",
  "found" : false
}
```

可以看出，found 属性值为 false，因为文档不存在，当然也就没有版本信息和 source 字段。

通过 HEAD 命令可以检查一个文档是否存在：

```
HEAD blog/article/1
```

如果文档存在，返回“200 – OK”，反之返回“404 - Not Found”。

如果想根据 id 一次获得多个文档，可以使用 Multi GET API 根据索引名、类型名、id（或者路由）一次获取多个文档，返回一个文档数组。举例如下：

```
GET _mget
{
    "docs" : [
        {
            "_index" : "blog",
            "_type" : "article",
            "_id" : "1"
        },
        {
            "_index" : "twitter",
            "_type" : "tweet",
            "_id" : "2"
        }
    ]
}
```

如果是同一个 index 下的不同 type，可以简写如下：

```
GET blog/_mget
{
  "docs": [
    {
      "_type": "typeA",
      "_id": "1"
    },
    {
      "_type": "typeB",
      "_id": "2"
    }
  ]
}
```

如果 index 和 type 都相同，可以简写为：

```
GET blog/article/_mget
{
    "docs" : [
```

```
        { "_id" : "1"},
        {"_id" : "2"}
    ]
}
```

进一步简化：

```
GET blog/article/_mget
{
    "ids" : ["1", "2"]
}
```

5.2.3　更新文档

文档被索引之后，如果要更新，那么 Elasticsearch 内部首先要找到这个文档，删除旧的文档内容执行更新，更新完后再索引最新的文档。下面以具体例子介绍 Update API。

首先，索引一条文档，命令如下：

```
PUT test/type1/1
{
    "counter" : 1,
    "tags" : ["red"]
}
```

现在对其进行更新操作，把 counter 字段的值加上 4，更新命令如下：

```
POST test/type1/1/_update
{
    "script" : {
        "inline": "ctx._source.counter += params.count",
        "lang": "painless",
        "params" : {
            "count" : 4
        }
    }
}
```

执行完以上命令之后，counter 字段的值将变为 5。命令中 inline 是执行的脚本，ctx 是脚本语言中的一个执行对象，painless 是 Elasticsearch 内置的一种脚本语言，params 是参数集合。上述命令的自然语言描述如下：使用 painless 脚本更新文档，通过 ctx 获取_source 再修改 counter 字段，counter 字段等于原值加上 count 参数的值。

ctx 对象除了可以访问_source 之外，还可以访问_index、_type、_id、_version、_routing、_parent 等字段。

tags 字段的取值为数组类型，先对其增加一个值，命令如下：

```
POST test/type1/1/_update
{
```

```
    "script" : {
        "inline": "ctx._source.tags.add(params.tag)",
        "lang": "painless",
        "params" : {
            "tag" : "blue"
        }
    }
}
```

如果想给文档新增一个字段，可以执行以下命令：

```
POST test/type1/1/_update
{
    "script" : "ctx._source.new_field = \"value_of_new_field\""
}
```

同样也可以移除一个字段：

```
POST test/type1/1/_update
{
    "script" : "ctx._source.remove(\"new_field\")"
}
```

删除 tags 数组中含有 red 的文档，命令如下（其中 ctx.op 等于 delete，表示删除该文档；ctx.op 等于 none，表示不执行任何操作）：

```
POST test/type1/1/_update
{
    "script" : {
        "inline": "if (ctx._source.tags.contains(params.tag))
                  {ctx.op = \"delete\" } else { ctx.op = \"none\" }",
        "lang": "painless",
        "params" : { "tag" : "red" }
    }
}
```

此外，更新文档还有一个 upsert 操作，其作用是处理待更新的文档不存在的情况。如果文档不存在，upsert 会新建一个文档，文档存在，则正常执行 script 脚本。下面的命令中，如果文档 test/type1/1 存在且有 counter 字段，则会执行 script 中的内容，counter 的值会增加 4；若不存在文档 test/type1/1，则会新建文档 test/type1/1 并新增一个字段 counter。

```
POST test/type1/1/_update
{
    "script" : {
        "inline": "ctx._source.counter += params.count",
        "lang": "painless",
        "params" : {"count" : 4}
```

```
    },
    "upsert" : {
        "counter" : 1
    }
}
```

5.2.4 查询更新

Elasticsearch 支持条件查询更新文档，使用的是 Update By Query API。下面给出一个查询更新的例子，对 title 中包含 git 关键字的文档增加一个 category 字段，命令如下：

```
POST blog/_update_by_query
{
  "script": {
    "inline": "ctx._source.category = params.category",
    "lang": "painless",
    "params": { "category":"git" }
  },
  "query": {
    "term": { "title": "git"}
  }
}
```

5.2.5 删除文档

Delete API 允许基于指定的 id 从索引库中删除一个文档，删除文档 blog/article/1 的命令如下：

```
DELETE blog/article/1
```

删除成功后，Elasticsearch 返回信息中会给出删除操作的响应结果，格式如下：

```
{
  "found": true,
  "_index": "blog",
  "_type": "article",
  "_id": "1",
  "_version": 1,
  "result": "deleted",
  "_shards": {
    "total": 1,
    "successful": 1,
    "failed": 0
  }
}
```

如果在索引文档时指定了路由，删除时也可以增加路由参数，命令如下：

```
DELETE blog/article/1?routing=user123
```

请注意，如果执行删除操作时路由值不正确，会导致文档删除失败。当映射的_routing 被设定为 required 且没有指定的路由值时，执行删除操作将抛出路由缺失异常并拒绝该请求。

5.2.6　查询删除

Delete By Query API 可以实现根据查询条件删除文档，在查询删除执行期间，依次执行多个搜索请求，以便找到要删除的所有匹配文档。每次发现一批文档时，执行相应的批量请求以删除所有这些文档。

通过 Delete By Query API 删除 title 中含有 Java 关键字的文档，命令如下：

```
POST blog/_delete_by_query
{
  "query":{
    "term": {
       "title":"hibernate"
    }
  }
}
```

删除一个 type 下的所有文档，命令如下：

```
POST blog/csdn/_delete_by_query
{
  "query": {
    "match_all": {}
  }
}
```

5.2.7　批量操作

如果文档数量非常大，一个一个操作文档显然不太符合实际，Elasticsearch 提供了文档的批量操作机制，通过 Bulk API 可以执行批量索引、批量删除、批量更新等操作。就像 mget 允许一次性检索多个文档一样，Bulk API 允许使用单一请求来实现多个文档的 create、index、update 或 delete。Bulk API 的使用方法如下：

步骤 01 创建一个 JSON 文件。

步骤 02 文件中写入多个请求操作，请求的格式如下：

```
action_and_meta_data\n
optional_source\n
....
action_and_meta_data\n
optional_source\n
```

步骤 03 执行操作

```
curl -XPOST 'localhost:9200/indexname/_bulk?pretty' --data-binary
@accounts.json
```

下面对请求的格式做详细的解释。每一行的结尾处都必须有换行字符"\n"，最后一行也要有，换行符可以有效地分隔每行。另外，这些行里不能包含非转义字符，以免干扰数据解析。

action_and_meta_data 行指定了将要在哪个文档中执行什么操作，其中 action 必须是 index、create、update 或者 delete，metadata 需要指明需要被操作文档的_index、_type 以及_id。

例如创建文档命令就可以这样填写：

```
{ "index":  { "_index": "blog", "_type": "article", "_id": "1" }}
{ "title":    "blog  title " }
```

也可以这样写：

```
{ "create": { "_index": " blog ", "_type": " article ", "_id": "1" }}
{ "title":    "blog  title " }
```

区别是如果文档 blog / article /1 已存在，create 会创建失败，index 不会。

如果不设置文档 id 的话，Elasticsearch 会自动创建，下面这种省略 id 的写法也是可行的：

```
{ "index": { "_index": " blog ", "_type": " article " }}
{ "title":    "blog  title " }
```

delete 请求格式如下：

```
{ "delete": { "_index": "website", "_type": "blog", "_id": "123" }}
```

新建一个 blog.json，写入以下内容，包含索引文档请求、更新文档请求和删除文档请求：

```
{ "delete": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "My first blog post" }
{ "index":  { "_index": "website", "_type": "blog" }}
{ "title":    "My second blog post" }
{ "update": { "_index": "website", "_type": "blog", "_id": "123",
               "_retry_on_conflict" : 3} }
{ "doc" : {"title" : "My updated blog post"} }
```

执行命令：

```
curl -XPOST "http://localhost:9200/website/_bulk?pretty" --data-binary
@blog.json
```

Elasticsearch 响应包含一个 items 数组，它罗列了每一个请求的结果，结果的顺序与请求的顺序相同：

```
{
  "took" : 110,
  "errors" : false,
  "items" : [
    {
      "delete" : {
        "found" : false,
        "_index" : "website",
        "_type" : "blog",
        "_id" : "123",
        "_version" : 1,
        "result" : "not_found",
        "_shards" : {
          "total" : 2,
          "successful" : 1,
          "failed" : 0
        },
        "status" : 404
      }
    },
    {
      "create" : {
        "_index" : "website",
        "_type" : "blog",
        "_id" : "123",
        "_version" : 2,
        "result" : "created",
        "_shards" : {
          "total" : 2,
          "successful" : 1,
          "failed" : 0
        },
        "created" : true,
        "status" : 201
      }
    },
    {
      "index" : {
        "_index" : "website",
        "_type" : "blog",
        "_id" : "AV4OkH3vpTAtvdgRj4Kv",
        "_version" : 1,
        "result" : "created",
```

```
        "_shards" : {
          "total" : 2,
          "successful" : 1,
          "failed" : 0
        },
        "created" : true,
        "status" : 201
      }
    },
    {
      "update" : {
        "_index" : "website",
        "_type" : "blog",
        "_id" : "123",
        "_version" : 3,
        "result" : "updated",
        "_shards" : {
          "total" : 2,
          "successful" : 1,
          "failed" : 0
        },
        "status" : 200
      }
    }
  ]
}
```

使用 Bulk 操作需要注意一次提交请求文件的大小，整个批量请求需要被加载到接受请求节点的内存里，所以请求越大，给其他请求可用的内存就越小。有一个最佳的 Bulk 请求大小，超过这个大小，性能不再提升而且可能降低。最佳大小不是一个固定的数字，它完全取决于服务器的硬件、文档的大小和复杂度以及索引和搜索的负载。幸运的是，这个最佳点（sweetspot）还是容易找到的，可以采用如下方式：试着批量索引标准的文档，随着大小的增长，当性能开始降低，说明你每个批次的大小太大了。开始的数量可以在 1000~5000 个文档之间，如果你的文档非常大，可以使用较小的批次。通常着眼于你请求批次的物理大小是非常有用的。一千个 1KB 的文档和一千个 1MB 的文档大不相同。一个好的批次最好保持在 5~15MB 之间。

5.2.8　版本控制

当我们使用 Elasticsearch 的 API 进行文档更新的时候整个过程如下：首先读取源文档，对原文档进行更新操作，更新操作执行完成以后再重新索引整个文档。不论执行多少次更新，最后保存在 Elasticsearch 中的是最后一次更新后的文档。但是如果有两个线程同时修改一个文档，这时候就会发生冲突。

如图 5-3 所示，假设一件商品的数量是 100，用户 A 买走了一件，与此同时，用户 B 也执

行了下单操作，但是 B 并不知道 A 也在下单，最终会返回商品还有 99 件，这样会造成系统中显示的商品数量比实际数量要多，这种情况在商业系统中是不能容忍的。

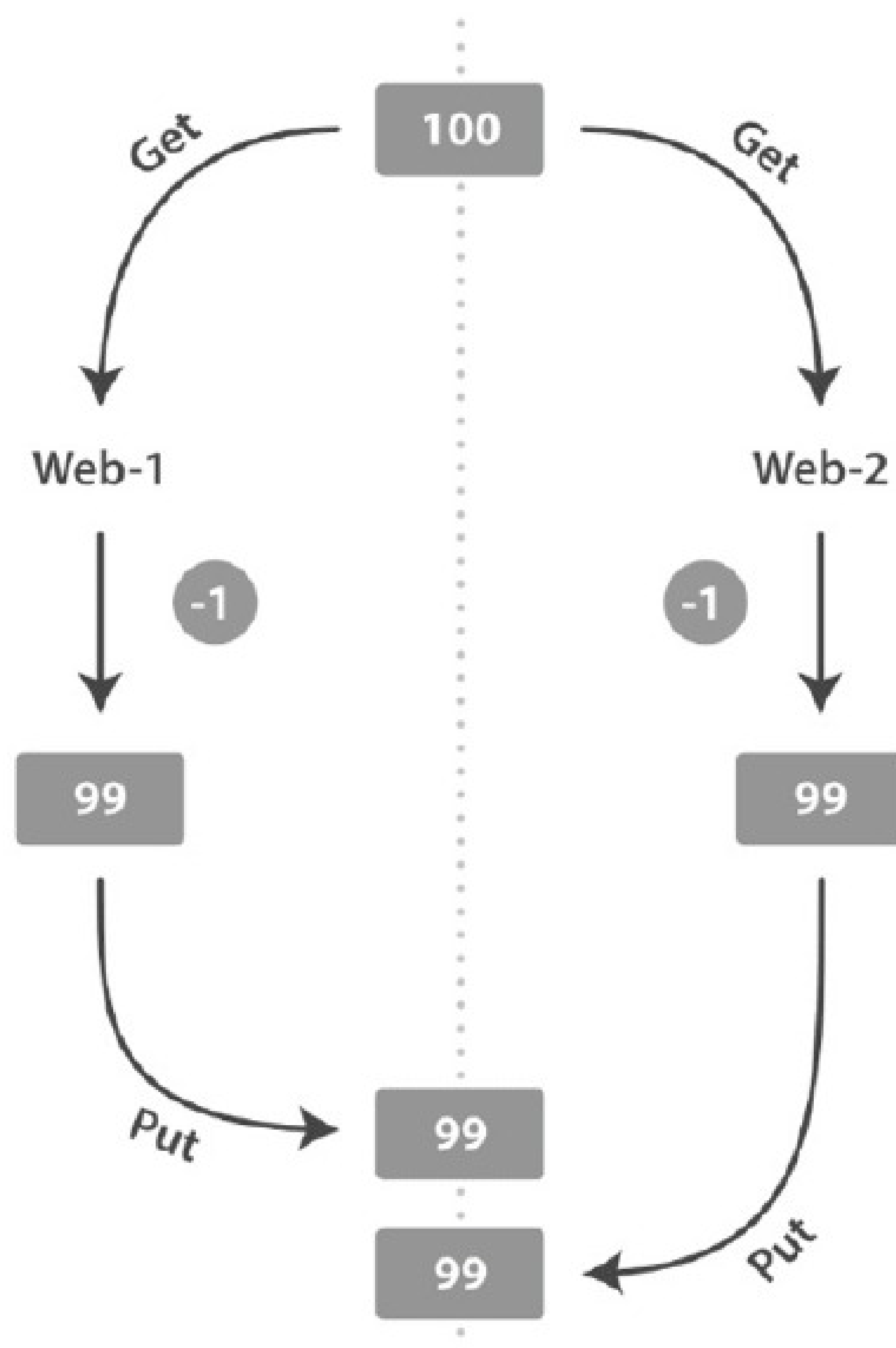


图 5-3　多线程并发操作

不对并发冲突进行控制，有时候不会造成太大影响。一个常见的场景是使用关系型数据库做主存储，使用 Elasticsearch 做检索引擎，因关系型数据库中有保护数据一致性的机制，只需要在 Elasticsearch 中使数据可搜索；或者偶尔的数据丢失不会对业务造成影响，比如对上亿条数据做数据分析，丢失一两条数据对整个结果不会有太大影响。但在要求数据强一致性的业务中缺少并发控制就会带来隐患，仍然需要控制。

那么如何进行并发控制？在数据库领域，确保在并发更新时数据不会丢失的方法主要有以下两种：

1. 悲观锁控制

顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，屏蔽一切有可能违反数据完整性的操作。悲观锁控制在关系型数据库中被广泛应用，这种方式假定冲突是有可能发生的。如果有线程对数据进行修改就对数据进行锁定，其他线程想要访问需要等待当前锁定释放，这样可确保同一时刻最多只有一个线程访问数据。传统的关系型数据库就用到了很多这种锁机制，比如行锁、表锁、读锁、写锁等。

2. 乐观锁控制

顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，假定不会发生并发访问冲突，对数据资源不会锁定，只有在数据提交操作时检查是否违反数据完整性。Elasticsearch 使用的就是乐观锁机制，乐观锁适用于读操作比较多的应用类型，可省去锁开销，可以提高吞吐量。

Elasticsearch 是一个分布式系统，当文档被创建、更新、删除，新版本的文档必须要复制

到集群中的其他节点。Elasticsearch 也是异步并发的，这意味着复制请求会被并行发送，也意味着请求不是按顺序到达的，Elasticsearch 需要一种方式确保旧版本的文档不会覆盖较新版本的文档。

我们在前面对文档进行索引时提到，文档每被修改一次，文档版本号会自增一次。Elasticsearch 使用_version 字段确保所有的更新都有序进行。如果一个低版本的文档在一个高版本的文档之后到达，那么旧版本的文档会被忽略。Elasticsearch 的文档版本控制机制主要有内部版本控制和外部版本控制，内部版本控制机制要求每次操作请求，只有当版本号相等时才能操作成功，外部版本控制要求外部文档版本比内部文档版本高时才能更新成功。下面通过具体实例演示一遍。

首先，创建一个索引：

```
PUT website
```

添加一个文档：

```
PUT /website/blog/1
{
   "title": "My first blog entry",
   "text":  "Just trying this out..."
}
```

查看刚索引的文档：

```
GET /website/blog/1
```

返回结果：

```
{
  "_index": "website",
  "_type": "blog",
  "_id": "1",
  "_version": 1,
  "found": true,
  "_source": {
    "title": "My first blog entry",
    "text": "Just trying this out..."
  }
}
```

可以看到，成功索引了一个文档，文档版本为 1，再更新该文档的标题：

```
POST website/blog/1/_update
{
  "script": "ctx._source.title=\"Update My first blog\""
}
```

此时文档版本已变为 2。如果再查看该文档并指定文档版本为 1，会发生版本冲突异常，带版本号的获取文档命令如下：

```
GET website/blog/1?version=1
```

在更新文档时可以指定外部文档的版本号，如果外部版本不高于当前文档版本，同样会发生异常。只有外部版本比当前文档版本高，更新操作才能成功执行。

```
PUT website/blog/1?version=5&version_type=external
{
  "title": "My  blog entry ",
  "text": " Starting to get the hang of this..."
}
```

5.2.9　路由机制

Elasticsearch 是一个分布式系统，当索引一个文档时文档会被存储到 master 节点上的一个主分片上。那么 Elasticsearch 是如何知道文档属于哪个分片的呢？再有当你创建一个新文档，Elasticsearch 是如何知道应该存储在分片 1 还是分片 2 上？要想回答这些问题，就需要了解 Elasticsearch 的路由机制。Elasticsearch 的路由机制即是通过哈希算法，将具有相同哈希值的文档放置到同一个主分片中，分片位置计算方法:

```
shard = hash(routing) % number_of_primary_shards
```

routing 值可以是一个任意字符串，Elasticsearch 默认将文档的 id 值作为 routing 值，通过哈希函数根据 routing 字符串生成一个数字，然后除以主切片的数量得到一个余数（remainder），余数的范围永远是 0 到 number_of_primary_shards -1，这个数字就是特定文档所在的分片。这种算法基本上会保持所有数据在所有分片上的一个平均分布，而不会造成数据分配不均衡的情况。

也可以自定义 routing 值。默认的路由模式可以保证数据平均分布，文档分配算法对我们来说是透明的，很多时候性能也不是问题。自定义 routing 值在深入理解数据特征之后，能够带来很多使用上的方便和性能上的提升。

假设存在一个有 50 个分片的索引，在集群上执行一次查询的过程如下：

（1）查询请求首先被集群中的一个节点接收。
（2）接收到这个请求的节点，将这个查询广播到这个索引的每个分片上。
（3）每个分片执行完搜索查询并返回结果。
（4）结果在通道节点上合并、排序并返回给用户。

默认情况下，Elasticsearch 使用文档的 id 将文档平均分布于所有的分片上，这导致了 Elasticsearch 不能确定文档的位置，所以它必须将这个请求广播到所有的 50 个分片上去执行。主分片的数量在索引创建的时候是固定的，并且永远不能改变。因为如果分片的数量改变了，所有先前的路由值就会变成非法，文档相当于丢失了。使用自定义的路由模式，可以使查询更具目的性。你不必盲目地去广播查询请求，而是要告诉 Elasticsearch 你的数据在哪个分片上。

Elasticsearch 的 index、get、mget、delete、update 等文档 API 都可以接收一个 routing 参数，以索引文档为例，执行 index 操作时给文档设置一个 routing 参数，具有相同 routing 的文档会被分配到同一个分片上。示例如下：

```
PUT /website/blog/1?routing=user123
{
  "title": "My first blog entry",
  "text":  "Just trying this out..."
}
```

上述命令索引了一条文档到 Elasticsearch 中，并指定 routing 为 user123，代表 user123 发布的所有博客。如果想要查询 user123 发布了哪些博客，可以通过 routing 值进行过滤，这样可以避免 Elasticsearch 向所有分片都发送查询请求，大大减少系统的资源。带 routing 参数的查询命令如下：

```
GET /myblog/_search?routing=user123
```

需要注意的是，可以为文档指定多个路由值，路由值之间使用逗号隔开。

使用自定义 routing 值也会造成一些潜在的问题，比如 user123 本身的文档就非常多，有数十万个，而其他大多数的用户只有几个文档，这样的话就会导致 user123 所在的分片较大，出现数据偏移的情况，特别是多个这样的用户处于同一分片的时候现象会更明显。具体的使用还是要结合实际的应用场景来选择。

5.3　映射详解

映射也就是 Mapping，用来定义一个文档以及其所包含的字段如何被存储和索引，可以在映射中事先定义字段的数据类型、分词器等属性。在关系型数据库中创建数据表时会设置字段的类型，如下 SQL 语句所示，创建一个 Student 表：

```
CREATE TABLE Student(
   id INT NOT NULL AUTO_INCREMENT,
   name VARCHAR(10) NOT NULL,
   PRIMARY KEY (stuid)
);
```

Elasticsearch 创建索引时同样可以设置字段的属性，作用是使索引的配置更加灵活和完善，可以在 Mapping 中设置字段的类型、字段的权重等信息。

5.3.1　映射分类

映射可分为动态映射和静态映射。在关系型数据库中写入数据之前首先要建表，在建表语句中声明字段的属性，在 Elasticsearch 中则不必如此，Elasticsearch 最重要的功能之一就是让你尽可能快地开始探索数据，文档写入 Elasticsearch 中，它会根据字段的类型自动识别，这种机制称为动态映射，而静态映射则是写入数据之前对字段的属性进行手工设置。

5.3.2　动态映射

动态映射是一种偷懒的方式，可直接创建索引并写入文档，文档中字段的类型是 Elasticsearch 自动识别的，不需要在创建索引的时候设置字段的类型。在实际项目中，如果遇到的业务在导入数据之前不确定有哪些字段，也不清楚字段的类型是什么，使用动态映射非常合适。当 Elasticsearch 在文档中碰到一个以前没见过的字段时，它会利用动态映射来决定该字段的类型，并自动把该字段添加到映射中，根据字段的取值自动推测字段类型的规则见表 5-1。

表5-1　Elasticsearch自动推测字段类型的规则

JSON 格式的数据	自动推测的字段类型
null	没有字段被添加
true or false	boolean 类型
浮点类型数字	float 类型
数字	long 类型
JSON 对象	object 类型
数组	由数组中第一个非空值决定
string	有可能是 date 类型（开启日期检测）、double 或 long 类型、text 类型、keyword 类型

下面举一个例子认识动态 Mapping，在 Elasticsearch 中创建一个新的索引并查看它的 Mapping，命令如下：

```
PUT books
GET books/_mapping
```

此时 books 索引的 Mapping 是空的，返回结果如下：

```
{
  "books": {
    "mappings": {}
  }
}
```

再往 books 索引中写入一条文档，命令如下：

```
PUT books/it/1
{
  "id":1,
  "publish_date":"2017-06-01",
  "name":"master Elasticsearch"
}
```

文档写入完成以后，再次查看 Mapping，返回结果如下：

```
{
  "books": {
    "mappings": {
      "it": {
        "properties": {
          "id": { "type": "long"},
          "name": {
            "type": "text",
            "fields": {
              "keyword": {
                "type": "keyword",
                "ignore_above": 256
              }
            }
          },
          "publish_date": { "type": "date"}
        }
      }
    }
  }
}
```

id、publish_date、name 三个字段分别被推测为 long 类型、date 类型和 text 类型，这就是动态 Mapping 的功劳。

使用动态 Mapping 要结合实际业务需求来综合考虑，如果将 Elasticsearch 当作主要的数据存储使用，并且希望出现未知字段时抛出异常来提醒你注意这一问题，那么开启动态 Mapping 并不适用。在 Mapping 中可以通过 dynamic 设置来控制是否自动新增字段，接受以下参数：

- true　默认值为 true，自动添加字段。
- false　忽略新的字段。
- strict　严格模式，发现新的字段抛出异常。

下面通过例子和实际操作来学习 dynamic 控制新增字段的方法。创建一个 books 索引并指定 Mapping，设置 it 类型下 dynamic 属性的取值为 strict，也就是说，it 类型下的文档中出现 Mapping 中没有定义的字段会抛出异常，命令如下:

```
PUT books
{
  "mappings": {
    "it": {
      "dynamic": "strict",
      "properties": {
```

```
          "title": {
            "type": "text"
          },
          "publish_date":{
            "type": "date"
          }
        }
      }
    }
}
```

写入三个文档：

```
PUT books/it/1
{
  "title": "master Elasticsearch",
   "publish_date":"2017-06-01"
}
PUT books/it/2
{
  "title": "master Elasticsearch"
}
PUT books/it/3
{
  "title": "master Elasticsearch",
   "publish_date":"2017-06-01",
   "author":"Tom"
}
```

文档 books/it/1 和 books/it/2 会创建成功，books/it/3 中出现了新的字段 author，该字段在 Mapping 中并没有定义，会抛出 strict_dynamic_mapping_exception 异常。

5.3.3 日期检测

当 Elasticsearch 碰到一个新的字符串类型的字段时，它会检查这个字符串是否包含一个可识别的日期，比如 2014-01-01。如果看起来像日期，那么它会被识别为一个 date 类型的字段，否则会将它作为 string 字段进行添加。这种自动检测机制有时会导致一些问题，假设一种情况，比如索引一份这样的文档到 Elasticsearch 中：

```
{ "note": "2014-01-01" }
```

如果 note 字段第一次被发现，那么根据规则它会被作为 date 字段添加。但是如果下一份文档是这样的：

```
{ "note": "Logged out" }
```

这时该字段显然不是日期类型，但是已经太迟了。该字段的类型已经是日期类型的字段了，因此这会导致一个异常被抛出。可以通过在根对象上将 date_detection 设置为 false 来关闭日期检测，命令如下：

```
PUT /my_index
{
    "mappings": {
        "my_type": {
            "date_detection": false
        }
    }
}
```

有了以上的映射，一个字符串总是会被当作 string 类型。如果需要新增一个 date 类型的字段，需要手动添加。

5.3.4　静态映射

静态映射是在创建索引时手工指定索引映射，和 SQL 中在建表语句中指定字段属性类似。相比动态映射，通过静态映射可以添加更详细、更精准的配置信息，例子如下：

```
PUT my_index
{
  "mappings": {
    "user": {
      "_all":       { "enabled": false  },
      "properties": {
        "title":    { "type": "text"  },
        "name":     { "type": "text"  },
        "age":      { "type": "integer" }
      }
    },
    "blogpost": {
      "_all":       { "enabled": false  },
      "properties": {
        "title":    { "type": "text"  },
        "body":     { "type": "text"  },
        "user_id":  {
          "type":   "keyword"
        },
        "created":  {
          "type":   "date",
```

```
            "format": "strict_date_optional_time||epoch_millis"
          }
        }
      }
    }
  }
```

5.3.5　字段类型

Elasticsearch 字段类型主要有核心类型、复合类型、地理类型和特殊类型，具体分类见表 5-2。

表5-2　Elasticsearch字段类型

一级分类	二级分类	具体类型
核心类型	字符串类型	string、text、keyword
	数字类型	long、integer、short、byte、double、float、half_float、scaled_float
	日期类型	date
	布尔类型	boolean
	二进制类型	binary
	范围类型	range
复合类型	数组类型	array
	对象类型	object
	嵌套类型	nested
地理类型	地理坐标	geo_point
	地理图形	geo_shape
特殊类型	IP 类型	ip
	范围类型	completion
	令牌计数类型	token_count
	附件类型	attachment
	抽取类型	percolator

1. string

Elasticsearch 5.X 之后的字段类型不再支持 string，由 text 或 keyword 取代。 如果仍使用 string，会给出警告。

2. text

如果一个字段是要被全文搜索的，比如邮件内容、产品描述、新闻内容，应该使用 text 类型。设置 text 类型以后，字段内容会被分析，在生成倒排索引以前，字符串会被分词器分成一个一个词项。text 类型的字段不用于排序，很少用于聚合（termsAggregation 除外）。

把 full_name 字段设为 text 类型的 Mapping 如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "full_name": {
          "type":  "text"
        }
      }
    }
  }
}
```

3. keyword

keyword 类型适用于索引结构化的字段，比如 email 地址、主机名、状态码和标签，通常用于过滤(比如查找已发布博客中 status 属性为 published 的文章)、排序、聚合。类型为 keyword 的字段只能通过精确值搜索到，区别于 text 类型。

4. 数字类型

数字类型支持 byte、short、integer、long、float、double、half_float 和 scaled_float，它们的取值范围见表 5-3。

表5-3　数字类型及其取值范围

类型	取值范围	类型	取值范围
long	-2^63 至 2^63-1	double	64 位双精度 IEEE 754 浮点类型
integer	-2^31 至 2^31-1	float	32 位单精度 IEEE 754 浮点类型
short	-32 768 至 32 767	half_float	16 位半精度 IEEE 754 浮点类型
byte	-128 至 127	scaled_float	缩放类型的浮点数

对于 float、half_float 和 scaled_float，-0.0 和+0.0 是不同的值，使用 term 查询查找-0.0 不会匹配+0.0，同样 range 查询中上边界是-0.0 不会匹配+0.0，下边界是+0.0 不会匹配-0.0。

对于数字类型的字段，在满足需求的情况下，要尽可能选择范围小的数据类型。比如，某个字段的取值最大值不会超过 100，那么选择 byte 类型即可。迄今为止，吉尼斯世界记录的人类的年龄的最大值为 134 岁，对于年龄字段，short 足矣。字段的长度越短，索引和搜索的效率越高。

处理浮点数时，优先考虑使用 scaled_float 类型。scaled_float 是通过缩放因子把浮点数变成 long 类型，比如价格只需要精确到分，price 字段的取值为 57.34，设置放大因子为 100，存储起来就是 5734。所有的 API 都会把 price 的取值当作浮点数，事实上 Elasticsearch 底层存储的是整数类型，因为压缩整数比压缩浮点数更加节省存储空间。

数字类型配置映射的例子如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "number_of_bytes": { "type": "integer" },
        "time_in_seconds": { "type": "float" },
        "price": {
          "type": "scaled_float",
          "scaling_factor": 100
        }
      }
    }
  }
}
```

5. date

JSON 中没有日期类型，所以在 Elasticsearch 中的日期可以是以下几种形式：

（1）格式化日期的字符串，如“2015-01-01”或“2015/01/01 12:10:30”。

（2）代表 milliseconds-since-the-epoch 的长整型数（epoch 指的是一个特定的时间：1970-01-01 00:00:00 UTC）。

（3）代表 seconds-since-the-epoch 的整型数。

Elasticsearch 内部会把日期转换为 UTC（世界标准时间），并将其存储为表示 milliseconds-since-the-epoch 的长整型数。日期格式可以自定义，如果没有自定义，默认格式如下：

```
"strict_date_optional_time||epoch_millis"
```

日期类型配置映射的例子如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {"date": { "type": "date" }}
    }
  }
}
```

写入 3 个文档：

```
PUT my_index/my_type/1
{ "date": "2015-01-01" }
```

```
PUT my_index/my_type/2
{ "date": "2015-01-01T12:10:30Z" }

PUT my_index/my_type/3
{ "date": 1420070400001 }
```

默认情况下，以上 3 个文档的日期格式都可以被解析，内部存储的是毫秒计时的长整型数。

6. boolean

如果一个字段是布尔类型，可接受的值为 true、false。Elasticsearch 5.4 版本以前，可以接受被解释为 true 或 false 的字符串和数字，5.4 版本以后只接受 true、false、“true”、“false”。

布尔类型配置映射的例子如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "is_published": { "type": "boolean" }
      }
    }
  }
}
```

写入三条文档：

```
POST my_index/my_type/1
{
  "is_published": true
}

POST my_index/my_type/2
{
  "is_published": "true"
}

POST my_index/my_type/3
{
  "is_published": false
}
```

执行以下搜索，文档 1 和文档 2 都可以被搜索：

```
GET my_index/_search
{
  "query": {
```

```
      "term": { "is_published": true}
    }
  }
```

7. binary

binary 类型接受 base64 编码的字符串，默认不存储（这里的存储是指 store 属性取值为 false）也不可搜索。

布尔类型配置映射的例子如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "name": { "type": "text" },
        "blob": { "type": "binary"}
      }
    }
  }
}
```

写入一条文档，其中 blob 的值为字符串“Some binary blob”的 Base64 编码：

```
PUT my_index/my_type/1
{
  "name": "Some binary blob",
  "blob": "U29tZSBiaW5hcnkgYmxvYg=="
}
```

8. array

Elasticsearch 没有专用的数组类型，默认情况下任何字段都可以包含一个或者多个值，但是一个数组中的值必须是同一种类型。例如：

（1）字符数组：[“one”，“two”]
（2）整型数组：[1，3]
（3）嵌套数组：[1，[2，3]],等价于[1，2，3]
（4）对象数组：[{ "name": "Mary"，"age": 12 }，{ "name": "John"，"age": 10 }]

动态添加数据时，数组的第一个值的类型决定整个数组的类型。混合数组类型是不支持的，比如：[1，“abc”]。数组可以包含 null 值，空数组[ ]会被当作 missing field 对待。

在文档中使用 array 类型不需要提前做任何配置，默认支持。例如写入一条带有数组类型的文档，命令如下：

```
PUT my_index/my_type/1
{
```

```
  "message": "some arrays in this document...",
  "tags":  [ "elasticsearch", "wow" ],
  "lists": [
    {
      "name": "prog_list",
      "description": "programming list"
    },
    {
      "name": "cool_list",
      "description": "cool stuff list"
    }
  ]
}
```

搜索 lists 字段下的 name，命令如下：

```
GET my_index/_search
{
  "query": {
    "match": {"lists.name": "cool_list"}
  }
}
```

9. object

JSON 本质上具有层级关系，文档包含内部对象，内部对象本身还包含内部对象，请看下面的例子：

```
PUT my_index/my_type/1
{
  "region": "US",
  "manager": {
    "age":     30,
    "name": {
      "first": "John",
      "last":  "Smith"
    }
  }
}
```

上面的文档中，整体是一个 JSON 对象，JSON 中包含一个 manager 对象，manager 对象又包含名为 name 的内部对象。写入到 Elasticsearch 之后，文档会被索引成简单的扁平 key-value 对，格式如下：

```
{
  "region":            "US",
  "manager.age":        30,
```

```
  "manager.name.first":  "John",
  "manager.name.last":  "Smith"
}
```

上面文档结构的显式映射如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "region": {
          "type": "keyword"
        },
        "manager": {
          "properties": {
            "age":  { "type": "integer" },
            "name": {
              "properties": {
                "first": { "type": "text" },
                "last":  { "type": "text" }
              }
            }
          }
        }
      }
    }
  }
}
```

10. nested

nested 类型是 object 类型中的一个特例，可以让对象数组独立索引和查询。Lucene 没有内部对象的概念，所以 Elasticsearch 将对象层次扁平化，转化成字段名字和值构成的简单列表。

使用 Object 类型有时会出现问题，比如文档 my_index/my_type/1 的结构如下：

```
PUT my_index/my_type/1
{
  "group" : "fans",
  "user" : [
    {
      "first" : "John",
      "last" :  "Smith"
    },
    {
```

```
      "first" : "Alice",
      "last" :  "White"
    }
  ]
}
```

user 字段会被动态添加为 Object 类型，最后会被转换为以下平整的形式：

```
{
  "group" :        "fans",
  "user.first" : [ "alice", "john" ],
  "user.last" :  [ "smith", "white" ]
}
```

user.first 和 user.last 扁平化以后变为多值字段，alice 和 white 的关联关系丢失了。执行以下搜索会搜索到上述文档：

```
GET /my_index/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": {"user.first": "Alice"}},
        {"match": {"user.last": "Smith"}}
      ]
    }
  }
}
```

事实上是不应该匹配的，如果需要索引对象数组并避免上述问题的产生，应该使用 nested 对象类型而不是 object 类型，nested 对象类型可以保持数组中每个对象的独立性。Nested 类型将数组中每个对象作为独立隐藏文档来索引，这意味着每个嵌套对象都可以独立被搜索，映射中指定 user 字段为 nested 类型：

```
PUT /my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "user": {
          "type": "nested"
        }
      }
    }
  }
}
```

再次执行上述查询语句，文档不会被匹配。

索引一个包含 100 个 nested 字段的文档实际上就是索引 101 个文档，每个嵌套文档都作为一个独立文档来索引。为了防止过度定义嵌套字段的数量，每个索引可以定义的嵌套字段被限制在 50 个。

11. geo point

geo point 类型用于存储地理位置信息的经纬度，可用于以下几种场景：

- 查找一定范围内的地理位置。
- 通过地理位置或者相对中心点的距离来聚合文档。
- 把距离因素整合到文档的评分中。
- 通过距离对文档排序。

指定 location 字段为 geo_point 类型，映射如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "location": {"type": "geo_point"}
      }
    }
  }
}
```

geo_point 字段接收以下 4 种类型的地理位置数据，分别介绍如下。

（1）经纬度坐标键值对

```
PUT my_index/my_type/1
{
  "text": "Geo-point as an object",
  "location": {
    "lat": 41.12,
    "lon": -71.34
  }
}
```

（2）字符串格式的地理坐标参数

```
PUT my_index/my_type/2
{
  "text": "Geo-point as a string",
  "location": "41.12,-71.34"
}
```

（3）地理坐标的哈希值

```
PUT my_index/my_type/3
{
  "text": "Geo-point as a geohash",
  "location": "drm3btev3e86"
}
```

（4）数组形式的地理坐标

```
PUT my_index/my_type/4
{
  "text": "Geo-point as an array",
  "location": [ -71.34, 41.12 ]
}
```

12. geo_shape

geo_point 类型可以存储一个坐标点，geo_shape 类型可以存储一块区域，比如矩形、三角形或者其他多边形。GeoJSON 是一种对各种地理数据结构进行编码的格式，对象可以表示几何、特征或者特征集合，支持点、线、面、多点、多线、多面等几何类型。GeoJSON 里的特征包含一个几何对象和其他属性，特征集合表示一系列特征。想了解更多关于 GeoJSON 的资料可参考《GeoJSON 格式规范说明》(http://www.oschina.net/translate/geojson-spec)。Elasticsearch 使用 GeoJSON 格式来表示地理形状，类型说明见表 5-4。

表 5-4　Elasticsearch 地理形状说明

GeoJSON 类型	Elasticsearch 类型	说明
Point	point	一个单独的经纬度坐标点
LineString	linestring	任意的线条，由两到多个点组成
Polygon	polygon	由 N+1 个点组成的封闭 N 边形
MultiPoint	multipoint	一组不连续但有可能相关联的点
MultiLineString	multilinestring	多条不关联的线
MultiPolygon	multipolygon	多个不关联的多边形
GeometryCollection	geometrycollection	几何对象的集合
N/A	envelope	由左上角坐标或右下角坐标确定的封闭矩形
N/A	circle	由圆心和半径确定的圆，默认单位为米

下面通过实例演示如何使用 geo_shape 类型。首先创建一个索引，映射中指定 location 字段为 geo_shape 类型，命令如下：

```
PUT geoshape
{
  "mappings": {
    "city":{
      "properties": {
```

```
        "location":{
          "type": "geo_shape"
        }
      }
    }
  }
}
```

写入一条由经纬度组成的点：

```
POST geoshape/city/1
{
  "location": {
    "type": "point",
    "coordinates": [
      -77.03653,
      38.897676
    ]
  }
}
```

写入一条由多个点组成的线：

```
POST geoshape/city/2
{
    "location" : {
        "type" : "linestring",
        "coordinates" : [[-77.03653, 38.897676], [-77.009051, 38.889939]]
    }
}
```

写入一条首尾封闭的多边形：

```
POST geoshape/city/3
{
    "location" : {
        "type" : "polygon",
        "coordinates" : [
            [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0,
              0.0] ]
        ]
    }
}
```

写入多个多边形：

```
POST geoshape/city/4
```

```
{
    "location" : {
        "type" : "polygon",
        "coordinates" : [
            [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0],
                [100.0, 0.0] ],
            [ [100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8],
                [100.2, 0.2] ]
        ]
    }
}
```

13. ip

ip 类型的字段用于存储 IPv4 或者 IPv6 的地址。在映射中指定字段为 ip 类型的映射和查询语句如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "ip_addr": {"type": "ip"}
      }
    }
  }
}

PUT my_index/my_type/1
{
  "ip_addr": "192.168.1.1"
}

GET my_index/_search
{
  "query": {
    "term": {
      "ip_addr": "192.168.0.0/16"
    }
  }
}
```

14. range

range 类型的使用场景包括网页中的时间选择表单、年龄范围选择表单等，range 类型支持的类型和取值范围见表 5-5。

表5-5　range类型及其取值范围

类　型	范　围
integer_range	-2^31 至 2^31-1
float_range	32-bit IEEE 754
long_range	-2^63 至 2^63-1
double_range	64-bit IEEE 754
date_range	64 位整数，毫秒计时

下面代码创建了一个 range_index 索引，expected_attendees 字段为 integer_range 类型，time_frame 字段为 date_range 类型。

```
PUT range_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "expected_attendees": {
          "type": "integer_range"
        },
        "time_frame": {
          "type": "date_range",
          "format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd||epoch_millis"
        }
      }
    }
  }
}
```

索引一条文档，expected_attendees 的取值为 10 到 20，time_frame 的取值是 2015-10-31 12:00:00 至 2015-11-01，命令如下。

```
PUT range_index/my_type/1
{
  "expected_attendees" : {
    "gte" : 10,
    "lte" : 20
  },
  "time_frame" : {
    "gte" : "2015-10-31 12:00:00",
    "lte" : "2015-11-01"
  }
}
```

15. token_count

token_count 用于统计字符串分词后的词项个数，本质上是一个整数型字段。举个例子，映射中指定 name 为 text 类型，增加 name.length 字段用于统计分词后词项的长度，类型为 token_count，分词器为标准分词器，命令如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "name": {
          "type": "text",
          "fields": {
            "length": {
              "type":    "token_count",
              "analyzer": "standard"
            }
          }
        }
      }
    }
  }
}
```

写入两条文档做测试，解析后第一条文档的 name.length 值为 2，第二条文档的 name.length 值为 3，命令如下。

```
PUT my_index/my_type/1
{
"name": "John Smith"
}

PUT my_index/my_type/2
{
"name": "Rachel Alice Williams"
}
```

搜索测试：

```
GET my_index/_search
{
  "query": {
    "term": {
```

```
        "name.length": 3
      }
    }
  }
```

5.3.6　元字段

元字段是映射中描述文档本身的字段，从大的分类上来看，主要有文档属性的元字段、源文档的元字段、索引的元字段、路由的元字段和自定义元字段，详细分类信息见表 5-6。

表5-6　Elasticsearch元字段分类

元字段分类	具体属性	作用
文档属性的元字段	_index	文档所属索引
	_uid	包含_type 和_id 的复合字段
	_type	文档的类型
	_id	文档 id
源文档的元字段	_source	文档的原始 JSON 字符串
	_size	_source 字段的大小
索引的元字段	_all	包含索引全部字段的超级字段
	_field_names	文档中包含非空值的所有字段
路由的元字段	_parent	指定文档间的父子关系
	_routing	将文档路由到特定分片的自定义路由值
自定义元字段	_meta	用于自定义元数据

1. _index

多索引查询时，有时候只需要在特定的索引名上进行查询，_index 字段提供了便利。_index 支持对索引名进行 term 查询、terms 查询、聚合分析、使用脚本和排序。

_index 是一个虚拟字段，不会真的加到 Lucene 索引中，可以对_index 进行 term、terms 查询（如 match、query_string、simple_query_string），但是不支持 prefix、wildcard、regexp 和 fuzzy 查询。举例如下，有 index_1 和 index_2 两个索引：

```
PUT index_1/my_type/1
{
  "text": "Document in index 1"
}

PUT index_2/my_type/2?refresh=true
{
  "text": "Document in index 2"
}
```

按照索引名进行搜索、排序和分组聚合：

```
GET index_1,index_2/_search
{
  "query": {
    "terms": {"_index": ["index_1", "index_2"] }
  },
  "aggs": {
    "indices": {
      "terms": {
        "field": "_index",
        "size": 10
      }
    }
  },
  "sort": [
    { "_index": { "order": "asc"}}
  ]
}
```

2. _type

每条被索引的文档都有一个_type 和_id 字段，可以根据_type 进行查询、聚合、脚本和排序。例子如下，my_index 索引下有两个 type：

```
PUT my_index/type_1/1
{
  "text": "Document with type 1"
}

PUT my_index/type_2/2?refresh=true
{
  "text": "Document with type 2"
}
```

对 type 进行搜索、排序和分组聚合：

```
GET my_index/_search
{
  "query": {
    "terms": {
      "_type": [ "type_1", "type_2" ]
    }
  },
  "aggs": {
    "types": {
      "terms": {
```

```
          "field": "_type",
          "size": 10
        }
      }
    },
    "sort": [{ "_type": { "order": "desc"}}]
  }
```

3. _id

每条被索引的文档都有一个_type 和_id 字段，_id 可以用于 term 查询、terms 查询、match 查询、query_string 查询、simple_query_string 查询，但是不能用于聚合、脚本和排序。查找 my_index 索引下 id 为 1 和 2 的文档，代码如下：

```
GET my_index/_search
{
  "query": {
    "terms": {"_id": [ "1", "2" ] }
  }
}
```

4. _uid

_uid 是_type 和_id 的组合，取值为{type}#{id}。和_type 一样，_uid 也可用于查询、聚合、脚本和排序。对_uid 进行查询、排序和分组聚合，命令如下：

```
GET my_index/_search
{
  "query": {
    "terms": {
      "_uid": [ "my_type#1", "my_type#2" ]
    }
  },
  "aggs": {
    "UIDs": {
      "terms": {
        "field": "_uid",
        "size": 10
      }
    }
  },
  "sort": [
    {
      "_uid": {
```

```
        "order": "desc"
      }
    }
  ]
}
```

5. _source

_source 存储文档的原始值，默认_source 字段是开启的，也可以在映射中通过 enabled 参数关闭：

```
PUT tweets
{
  "mappings": {
    "tweet": {
      "_source": {
        "enabled": false
      }
    }
  }
}
```

但是一般情况下不要关闭，因为 update、update_by_query、reindex，关键字高亮，数据备份，改变 mapping，升级索引。通过原始字段 debug 查询或者聚合等诸多操作都需要用到_source 字段中存储的文档原始值。

6. _size

_size 用于描述文档本身的字节大小，默认是不支持的。如果有统计文档大小的需求，需要安装 mapper-size 插件，安装命令如下：

```
bin/elasticsearch-plugin install mapper-size
```

然后在映射中开启_size 字段：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "_size": {
        "enabled": true
      }
    }
  }
}
```

索引文档后对_size 进行过滤：

```
GET my_index/_search
{
  "query": {
    "range": {
      "_size": {
        "gt":30
      }
    }
  }
}
```

7. _all

_all 字段是把其他字段拼接在一起的超级字段，所有的字段内容用空格分开，_all 字段会被解析和索引，但是不存储。当需要返回包含某个关键字的文档，但是不明确地搜某个字段的时候，可以对_all 字段进行搜索。例子如下：

```
PUT my_index/blog/1
{
  "title":    "Master Java",
  "content":     "learn java",
  "author": "Tom"
}
```

_all 字段包含:["Master", "Java", "learn", "Tom"]，对_all 字段进行搜索：

```
GET my_index/_search
{
  "query": {
    "match": {
      "_all": "Java"
    }
  }
}
```

8. _field_names

_field_names 字段用来存储文档中的所有非空字段的名字，这个字段常用于 exists 查询。例子如下：

```
PUT my_index/my_type/1
{
  "title": "This is a document"
}
PUT my_index/my_type/2?refresh=true
```

```
{
  "title": "This is another document",
  "body": "This document has a body"
}
```

搜索具有 body 字段的文档:

```
GET my_index/_search
{
  "query": {
    "terms": {
      "_field_names": [ "body" ]
    }
  }
}
```

结果会返回第二条文档，因为第一条文档只有 title 字段。

9. _parent

_parent 用于指定同一索引中文档的父子关系。下面例子中先在 mapping 中指定文档的父子关系，然后索引父文档，索引子文档时指定父 id，最后根据子文档查询父文档。

```
PUT my_index
{
  "mappings": {
    "my_parent": {},
    "my_child": {
      "_parent": {
        "type": "my_parent"
      }
    }
  }
}

PUT my_index/my_parent/1
{
  "text": "This is a parent document"
}

PUT my_index/my_child/2?parent=1
{
  "text": "This is a child document"
}

PUT my_index/my_child/3?parent=1&refresh=true
{
  "text": "This is another child document"
```

```
}

GET my_index/my_parent/_search
{
  "query": {
    "has_child": {
      "type": "my_child",
      "query": {
        "match": {
          "text": "child document"
        }
      }
    }
  }
}
```

10. _routing

路由参数，Elasticsearch 通过以下公式计算文档应该分到哪个分片上：

```
shard = hash(routing) % number_of_primary_shards
```

默认的_routing 值是文档的_id 或者_parent，通过_routing 参数可以设置自定义路由，在 5.2.9 小节中已经讲过。

在 Mapping 中指定 routing 为必需的：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "_routing": {
        "required": true
      }
    }
  }
}
```

指定文档的路由值为必需的之后，索引文档必须提供路由参数，否则会报错，测试命令如下：

```
PUT my_index/my_type/1
{
  "text": "No routing value provided"
}
```

5.3.7　映射参数

Elasticsearch 提供了足够多的映射参数对字段的映射进行参数设置，一些常用功能的实现，

比如字段的分词器、字段的权重、日期格式、检索模型的选择等都是通过映射参数来配置完成的，下面一一介绍各个参数的用法。

1. analyzer

analyzer 参数用于指定文本字段的分词器，对索引和查询都有效。分词器会把文本类型的内容转换为若干个词项，查询时分词器同样把查询字符串通过和索引时期相同的分词器或者其他分词器进行解析。以常用的 IK 中文分词器为例，对于 content 字段，analyzer 参数的取值为 ik_max_word，意味着 content 字段内容索引时和查询时都使用 ik_max_word 分词，配置映射的命令如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "content": {
          "type": "text",
          "analyzer": "ik_max_word"
        }
      }
    }
  }
}
```

search_analyzer

大多数情况下索引和搜索的时候应该指定相同的分词器，确保 query 解析以后和索引中的词项一致。但是有时候也需要指定不同的分词器，例如，使用 edge_ngram 过滤器实现自动补全。默认情况下查询会使用 analyzer 属性指定的分词器，但也可以被 search_analyzer 覆盖。例子如下：

```
PUT my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "autocomplete_filter": {
          "type": "edge_ngram",
          "min_gram": 1,
          "max_gram": 20
        }
      },
      "analyzer": {
        "autocomplete": {
```

```
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "autocomplete_filter"
          ]
        }
      }
    }
  },
  "mappings": {
    "my_type": {
      "properties": {
        "text": {
          "type": "text",
          "analyzer": "autocomplete",
          "search_analyzer": "standard"
        }
      }
    }
  }
}
```

text 字段使用 autocomplete 分词器进行索引，但是使用 standard 分词器进行搜索。索引一条文档：

```
PUT my_index/my_type/1
{
  "text": "Quick Brown Fox"
}
```

text 字段生成的倒排索引包含以下词项：

```
[ q, qu, qui, quic, quick, b, br, bro, brow, brown, f, fo, fox ]
```

normalizer

normalizer 参数用于解析前的标准化配置，比如把所有的字符转化为小写。下面的例子中 foo 字段的值在解析前使用自定义的 normalizer 把字符串标准化并转换为小写形式：

```
PUT index
{
  "settings": {
    "analysis": {
      "normalizer": {
        "my_normalizer": {
```

```
          "type": "custom",
          "char_filter": [],
          "filter": ["lowercase", "asciifolding"]
        }
      }
    }
  },
  "mappings": {
    "type": {
      "properties": {
        "foo": {
          "type": "keyword",
          "normalizer": "my_normalizer"
        }
      }
    }
  }
}
PUT index/type/1
{
  "foo": "BÀR"
}
PUT index/type/2
{
  "foo": "bar"
}
PUT index/type/3
{
  "foo": "baz"
}
POST index/_refresh
```

下面的查询会匹配文档 1 和 2，这是因为在索引和查询的时候都将 BAR 转换为了 bar：

```
GET index/_search
{
  "query": {
    "match": {
      "foo": "BAR"
    }
  }
}
```

4. boost

boost 字段用于设置字段的权重，比如，设置关键字出现在 title 字段的权重是出现在 content 字段中权重的两倍，其中 content 字段的默认权重是 1，mapping 如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "title": {
          "type": "text",
          "boost": 2
        },
        "content": {
          "type": "text"
        }
      }
    }
  }
}
```

同样，在查询时指定权重也是一样的：

```
POST _search
{
    "query": {
        "match" : {
            "title": {
                "query": "quick brown fox",
                "boost": 2
            }
        }
    }
}
```

推荐在查询时指定 boost。在索引期设置权重，如果不重新索引文档，权重无法修改；在查询时指定权重可以实现同样的效果，同时修改权重，使其更加灵活。

5. coerce

coerce 属性用于清除脏数据，默认值是 true。整型数字 5 有可能会被写成字符串“5”或者浮点数 5.0。coerce 属性可以用来清除脏数据，字符串和浮点数会被强制转换为整数。例子如下，其中映射中指定 number_one 和 number_two 都是 integer 类型，number_two 字段的 coerce 属性修改为 false。

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "number_one": {
          "type": "integer"
        },
        "number_two": {
          "type": "integer",
          "coerce": false
        }
      }
    }
  }
}
```

写入两条测试文档：

```
PUT my_index/my_type/1
{
  "number_one": "10"
}

PUT my_index/my_type/2
{
  "number_two": "10"
}
```

Mapping 中指定 number_one 字段是 integer 类型，虽然插入的数据类型是字符串，但依然可以插入成功，number_two 字段关闭了 coerce，因此插入失败。

6. copy_to

copy_to 参数用于自定义_all 字段，可以把多个字段的值复制到一个超级字段。下面的例子中把 title 字段和 content 字段的内容合并在一起生成一个 full_content。

```
PUT myindex
{
  "mappings": {
    "mytype": {
      "properties": {
        "title": {
          "type":    "text",
          "copy_to": "full_content"
        },
```

```
        "content": {
          "type":    "text",
          "copy_to": "full_content"
        },
        "full_content": {
          "type":    "text"
        }
      }
    }
  }
}
```

7. doc_values

doc_values 参数是为了加快排序、聚合操作，在建立倒排索引的时候，额外增加一个列式存储映射，是一种空间换时间的做法。默认是开启的，对于确定不需要聚合或者排序的字段可以关闭 doc_values 节省存储空间。

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "status_code": {
          "type":       "keyword"
        },
        "session_id": {
          "type":       "keyword",
          "doc_values":  false
        }
      }
    }
  }
}
```

注：text 类型不支持 doc_values。

8. dynamic

dynamic 属性用于检测新发现的字段，见 5.3.2 小节。

9. enabled

Elasticseaech 默认会索引所有的字段，而有些字段只需要存储，没有查询或者聚合的需求，这种情况下就可以使用 enabled 参数来控制。enabled 设为 false 的字段，Elasticseaech 会跳过字段内容，该字段只能从_source 中获取，但是不可以被搜索，字段可以是任意类型。例如，设置 name 字段的 enabled 取值为 false：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "name": {
          "enabled": false
        }
      }
    }
  }
}
```

enabled 参数还可以禁用映射，下面的命令会把 my_index 下的 my_type 类型禁用，再往 my_type 类型下写入文档，不会生成字段的 mapping 信息：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "enabled": false
    }
  }
}
```

10. fielddata

搜索要解决的问题是“包含查询关键词的文档有哪些？”，聚合恰恰相反，聚合要解决的问题是“文档包含哪些词项”，大多数字段在索引时都会生成 doc_values，text 字段除外。取而代之，text 字段在查询时会生成一个 fielddata 的数据结构，fielddata 在字段首次被聚合、排序或者使用脚本的时候生成。Elasticsearch 通过读取磁盘上的倒排记录表重新生成文档词项关系，最后在 Java 堆内存中排序。

text 字段的 fielddata 属性默认是关闭的，开启 fielddata 非常消耗内存。在你开启 text 字段以前，想清楚为什么要在 text 类型的字段上做聚合、排序操作，大多数情况下这么做是没有意义的。给 text 类型的字段开启 fielddata 的命令如下：

```
PUT my_index/_mapping/my_type
{
  "properties": {
    "my_field": {
      "type":     "text",
      "fielddata": true
    }
  }
}
```

11. format

Elasticsearch 中的 date 类型支持多种格式，format 参数就是用于指定日期格式的。下面的命令中给 date 字段设置了 3 种可接受的日期类型，2017-08-01、2017-08-01 12:10:10、1501560610 都是符合格式的日期值，更多的日期格式及其说明见表 5-7。

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "date": {
          "type":   "date",
          "format": " yyyy-MM-dd HH:mm:ss||yyyy-MM-dd||epoch_millis "
        }
      }
    }
  }
}
```

表5-7 日期格式

日期格式	含义	例子
epoch_millis	从 1970 年 1 月 1 日开始所经过的毫秒数	1505520000000（北京时间 2017/9/16 8:0:0）
epoch_second	从 1970 年 1 月 1 日开始所经过的秒数	1505520000（北京时间 2017/9/16 8:0:0）
date_optional_time 或 strict_date_optional_time	通用的 ISO 标准时间格式，日期是必须的，时间可选	2017-09-16
basic_date	完整的日期基本格式：yyyyMMdd	20170916
basic_date_time	带日期和时间的基本格式，日期和时间用 T 分隔：yyyyMMddTHHmmss.SSSZ	20170916T121000.826+0800
basic_date_time_no_millis	忽略毫秒的带日期和时间的基本格式：yyyyMMdd'T'HHmmssZ	20170916T121000+0800
basic_ordinal_date	基本日期，包括四位数年份和当前年份的具体天数：yyyyDDD	2017 年的第 100 天：2017100
basic_ordinal_date_time	四位数年份和当前年份的具体天数再加上时间： yyyyDDDTHHmmss.SSSZ	2017100T121000.826+0800
basic_ordinal_date_time_no_millis	四位数年份和当前年份的具体天数加上时间，忽略毫秒：yyyyDDDTHHmmssZ	2017100T121000+0800
basic_time	基本时间格式：HHmmss.SSSZ	121000.826+0800

（续表）

日期格式	含义	例子
basic_time_no_millis	忽略毫秒的基本时间格式: HHmmssZ	121000+0800
basic_t_time	带 T 标记的基本时间格式 THHmmss.SSSZ:	T121:00.826+0800
basic_t_time_no_millis	带 T 标记的基本时间格式，忽略毫秒: THHmmssZ	T121000+0800
basic_week_date 或 strict_basic_week_date	基本周格式日期: xxxxWwwe	2017 年第 12 周的第五天:2017W123
basic_week_date_time 或 strict_basic_week_date_time	基本周格式日期加上具体时间: xxxxWwweTHHmmss.SSSZ	2017W125T121000.826+0800
basic_week_date_time_no_millis 或 strict_basic_week_date_time_no_millis	基本周格式日期加上具体时间，忽略毫秒: xxxxWwweTHHmmssZ	2017W125T121000+0800
date 或 strict_date	日期格式:yyyy-MM-dd	2017-09-16
date_hour 或 strict_date_hour	日期格式加小时: yyyy-MM-ddTHH	2017-09-16T12
date_hour_minute 或 strict_date_hour_minute	日期格式加小时分钟: yyyy-MM-ddTHH:mm	2017-09-16T12:10
date_hour_minute_second 或 strict_date_hour_minute_second	日期格式加小时分钟和秒: yyyy-MM-ddTHH:mm:ss	2017-09-16T12:10:10
date_hour_minute_second_fraction 或 strict_date_hour_minute_second_fraction	日期格式加小时分钟秒和毫秒: yyyy-MM-ddTHH:mm:ss.SSS	2017-09-16T12:10:10.123
date_hour_minute_second_millis 或 strict_date_hour_minute_second_millis	日期格式加小时分钟秒和毫秒: yyyy-MM-ddTHH:mm:ss.SSS	2017-09-16T12:10:10.123
date_time 或 strict_date_time	日期加完整时间: yyyy-MM-ddTHH:mm:ss.SSSZZ	2016-07-15T12:58:17.136 +0800
date_time_no_millis 或 strict_date_time_no_millis	日期加完整时间，不带毫秒: yyyy-MM-ddTHH:mm:ss.SSSZZ	2017-09-16T12:10:10 .123+0800
hour 或 strict_hour	小时：HH	12
hour_minute 或 strict_hour_minute	小时加分钟:HH:mm	12:45
hour_minute_second 或 strict_hour_minute_second	小时分钟加秒：HH:mm:ss	12:45:20
hour_minute_second_fraction 或 strict_hour_minute_second_fraction	小时分钟秒加毫秒：HH:mm:ss.SSS	12:45:20.123
hour_minute_second_millis 或 strict_hour_minute_second_millis	小时分钟秒加毫秒：HH:mm:ss.SSS	12:45:20.123
ordinal_date 或 strict_ordinal_date	年份加一年中的第多少天:yyyy-DDD	2017-100
ordinal_date_time 或 strict_ordinal_date_time	年份加一年中的第多少天加具体时间:yyyy-DDDTHH:mm:ss.SSSZZ	2017-100T12:10:10 .123+0800

（续表）

日期格式	含义	例子
ordinal_date_time_no_millis 或 strict_ordinal_date_time_no_millis	年份加一年中的第多少天加具体时间不带毫秒: yyyy-DDDTHH:mm:ssZZ	2017-100T12:10:10+0800
time 或 strict_time	具体时间: HH:mm:ss.SSSZZ	12:10:10.123+0800
time_no_millis 或 strict_time_no_millis	具体时间不带毫秒: HH:mm:ssZZ	12:10:10+0800
t_time 或 strict_t_time	带 T 分隔符的时间: THH:mm:ss.SSSZZ	T12:10:10.123+0800
t_time_no_millis 或 strict_t_time_no_millis	带 T 分隔符的时间不带毫秒: THH:mm:ssZZ	T12:10:10+0800
week_date 或 strict_week_date	以周计时的日期: xxxx-Www-e	2017 年第 12 周的星期三:2017-W12-3
week_date_time 或 strict_week_date_time	以周计时的日期加时间: xxxx-Www-eTHH:mm:ss. SSSZZ	2017-W12-3T12:10:10.123+0800
week_date_time_no_millis 或 strict_week_date_time_no_millis	以周计时的日期加时间不带毫秒: xxxx-Www-eTHH:mm:ss. ZZ	2017-W12-3T12:10:10+0800
weekyear 或 strict_weekyear	年份:xxxx	2017
weekyear_week 或 strict_weekyear_week	年份加周数:xxxx-Www	2017-W12
weekyear_week_day 或 strict_weekyear_week_day	年份周数加天数: xxxx-Www-e	2017-W12-4
year 或 strict_year	年份:yyyy	2017
year_month 或 strict_year_month	年份加月:yyyy-MM	2017-09
year_month_day 或 strict_year_month_day	年月日:yyyy-MM-dd	2017-09-16

12. ignore_above

ignore_above 用于指定字段分词和索引的字符串最大长度，超过最大值的会被忽略，只用于 keyword 类型，例子如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "message": {
          "type": "keyword",
          "ignore_above": 20
        }
      }
    }
  }
```

```
}

PUT my_index/my_type/1
{
  "message": "short document"
}

PUT my_index/my_type/2
{
  "message": "another long document"
}

GET my_index/_search
{
  "size": 0,
  "aggs": {
    "messages": {"terms": {"field": "message"}}
  }
}
```

mapping 中指定了 ignore_above 字段的最大长度为 20，第一个文档中的 message 字段的字符数小于 20，第二个超过 20，聚合结果中只会返回第一个短文档：

```
{
  "aggregations": {
    "messages": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "short document",
          "doc_count": 1
        }
      ]
    }
  }
}
```

13. ignore_malformed

ignore_malformed 可以忽略不规则数据，对于 login 字段，有人可能填写的是 date 类型，也有人填写的是邮件格式。给一个字段索引不合适的数据类型会发生异常，进而导致整个文档索引失败。如果 ignore_malformed 参数设为 true，异常会被忽略，出异常的字段不会被索引，其他字段正常索引。

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "number_one": {
          "type": "integer",
          "ignore_malformed": true
        },
        "number_two": {
          "type": "integer"
        }
      }
    }
  }
}

PUT my_index/my_type/1
{
  "text":       "Some text value",
  "number_one": "foo"
}

PUT my_index/my_type/2
{
  "text":       "Some text value",
  "number_two": "foo"
}
```

上面的例子中 number_one 接受 integer 类型，ignore_malformed 属性设为 true，因此文档 1 中 number_one 字段虽然是字符串但依然能写入成功；number_two 只接受 integer 类型，默认 ignore_malformed 属性为 false，因此写入失败。

14. include_in_all

include_in_all 参数用于指定字段的值是否包含在_all 字段中，默认值为 true。如果需要把某个字段排除在_all 字段之外，可以把 include_in_all 参数的取值设为 false，命令如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "title": {
          "type": "text"
        },
```

```
        "content": {
          "type": "text"
        },
        "date": {
          "type": "date",
          "include_in_all": false
        }
      }
    }
  }
}
```

include_in_all 参数的取值对嵌套类型或对象类型的子字段都生效。

15. index

index 属性指定字段是否索引，不索引也就不可搜索，取值可以为 true 或者 false。

16. index_options

index_options 参数控制索引时存储哪些信息到倒排索引中，可取值见表 5-8。设置 text 字段存储文档编号、词项频率、词项位置、词项开始和结束的字符位置，映射如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "text": {
          "type": "text",
          "index_options": "offsets"
        }
      }
    }
  }
}
```

表5-8　index_options参数取值

参数	作用
docs	只存储文档编号，默认取值
freqs	存储文档编号和词项频率
positions	存储文档编号、词项频率、词项偏移位置，偏移位置可用于临近搜索和短语查询
offsets	文档编号、词项频率、词项的位置、词项开始和结束的字符位置都被存储，offsets 设为 true 会使用 Postings highlighter

17. fields

fields 参数可以让同一字段有多种不同的索引方式。比如一个文本类型的字段，可以使用

text 类型做全文检索，使用 keyword 类型做聚合和排序，映射如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "city": {
          "type": "text",
          "fields": {
            "raw": { "type":  "keyword"}
          }
        }
      }
    }
  }
}
```

18. norms

norms 参数用于标准化文档，以便查询时计算文档的相关性。norms 虽然对评分有用，但是会消耗较多的磁盘空间，如果不需要对某个字段进行评分，最好不要开启 norms。

19. null_value

值为 null 的字段不索引也不可以搜索，null_value 参数可以让值为 null 的字段显式的可索引、可搜索。例子如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "status_code": {
          "type":       "keyword",
          "null_value": "NULL"
        }
      }
    }
  }
}

PUT my_index/my_type/1
{
  "status_code": null
}
```

```
PUT my_index/my_type/2
{
  "status_code": []
}

GET my_index/_search
{
  "query": {
    "term": {
      "status_code": "NULL"
    }
  }
}
```

文档 1 可以被搜索到，因为 status_code 的值为 null，文档 2 不可以被搜索到，因为 status_code 为空数组，但是不是 null。

20. position_increment_gap

为了支持近似或者短语查询，text 类型的字段被解析的时候会考虑词项的位置信息。举例，一个字段的值为数组类型：

```
 "names": [ "John Abraham", "Lincoln Smith"]
```

为了区别第一个字段和第二个字段，Abraham 和 Lincoln 在索引中有一个间距，默认是 100。例子如下，这时查询“Abraham Lincoln”是查不到的：

```
PUT my_index/groups/1
{
    "names": [ "John Abraham", "Lincoln Smith"]
}

GET my_index/groups/_search
{
    "query": {
        "match_phrase": {
            "names": {
                "query": "Abraham Lincoln"
            }
        }
    }
}
```

指定间距大于 100 时，就可以查询到上述文档：

```
GET my_index/groups/_search
{
```

```
    "query": {
      "match_phrase": {
        "names": {
          "query": "Abraham Lincoln",
          "slop": 101
        }
      }
    }
}
```

在 mapping 中通过 position_increment_gap 参数指定间距，命令如下：

```
PUT my_index
{
  "mappings": {
    "groups": {
      "properties": {
        "names": {
          "type": "text",
          "position_increment_gap": 0
        }
      }
    }
  }
}
```

21. properties

类型的映射、普通字段、object 类型和 nested 类型的字段都称为 properties（属性），这些属性可以为任意的数据类型，包括 object 和 nested 类型。属性可以通过以下方式加入：

- 在创建索引时明确地定义它们。
- 在使用 PUT mapping API 添加或更新映射类型时明确地定义它们。
- 索引包含新字段的文档时动态地加入。

22. similarity

similarity 参数用于指定文档评分模型，参数有三个：

- BM25　Elasticsearch 和 Lucene 默认的评分模型。
- classic　TF/IDF 评分。
- boolean　布尔模型评分。

```
PUT my_index
{
  "mappings": {
    "my_type": {
```

```
        "properties": {
          "default_field": {
            "type": "text"
          },
          "classic_field": {
            "type": "text",
            "similarity": "classic"
          },
          "boolean_sim_field": {
            "type": "text",
            "similarity": "boolean"
          }
        }
      }
    }
  }
```

default_field 自动使用 BM25 评分模型，classic_field 使用 TF/IDF 经典评分模型，boolean_sim_field 使用布尔评分模型。

23. store

默认情况下，字段是被索引的，也可以搜索，但是不存储，这也没关系，因为_source 字段里保存了一份原始文档。在某些情况下，store 参数有意义，比如一个文档里有 title、date 和超大的 content 字段，如果只想获取 title 和 date，可以这样配置：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "title": {
          "type": "text",
          "store": true
        },
        "date": {
          "type": "date",
          "store": true
        },
        "content": {
          "type": "text"
        }
      }
    }
  }
}
```

24. term_vector

词向量包含了文本被解析以后的以下信息：

- 词项集合。
- 词项位置。
- 词项的起始字符映射到原始文档中的位置。

term_vector 参数的取值见表 5-9。

表5-9　term_vector参数的取值

参数取值	含义
no	默认值，不存储词向量
yes	只存储词项集合
with_positions	存储词项和词项位置
with_offsets	词项和字符偏移位置
with_positions_offsets	存储词项、词项位置、字符偏移位置

5.3.8　映射模板

通过动态映射模板，可以在 mapping 之上拥有对新字段的完整控制，甚至可以根据字段的名称来设置映射。每个模板都有一个名字，用来描述这个模板做了什么。同时它有一个映射用来指定具体的映射信息，还有至少一个参数（比如 match）用来规定对于什么字段需要使用该模板。模板的匹配有先后，只有第一个匹配的模板会被使用。下面的例子中增加了一个名为 longs_as_strings 的映射模板，如果字段名称以 long_开头，字符串转为 long 类型，映射如下：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "dynamic_templates": [
        {
          "longs_as_strings": {
            "match_mapping_type": "string",
            "match":   "long_*",
            "unmatch": "*_text",
            "mapping": {
              "type": "long"
            }
          }
        }
      ]
    }
  }
}
```

写入一条测试文档：

```
PUT my_index/my_type/1
{
  "long_num": "5",
  "long_text": "foo"
}
```

long_num 字段会被解析为 long 类型，long_text 仍为默认的字符串类型，match_mapping_type 允许只对特定类型的字段使用模板，正如标准动态映射规则那样，比如 string、long 等类型，match 参数中指定可以匹配的字段名的规则，unmatch 参数指定不匹配的字段名规则。另外，path_match 参数用于匹配字段的完整路径，比如 address.*.name 可以匹配如下字段：

```
{
    "address": {
        "city": {
            "name": "New York"
        }
    }
}
```

5.4　本章小结

本章介绍了 Elasticsearch 的基础知识，主要包括如何在 Elasticsearch 中进行索引管理、文档管理以及如何配置映射，这些都是为存储文档做铺垫。同时也总结了一些经验和技巧，为后续的学习打下基础。

第6章

Elasticsearch 搜索详解

本章学习要点：

- ✱ 配置 Elasticsearch 中文分词器
- ✱ 如何实现基本查询搜索
- ✱ 如何使用复合查询
- ✱ 如何使用过滤器
- ✱ 如何实现搜索高亮
- ✱ 文档的父子关系与嵌套搜索

6.1 搜索机制

Elasticsearch 的核心功能是搜索，有了前面的基础，可以合理地把文档索引到 Elasticsearch 之中，本章介绍 Elasticsearch 提供的丰富的搜索 API 及其用法。

为了更好地理解搜索机制，首先从宏观上认识搜索流程。图 6-1 描述了一条文档从索引到 Elasticsearch 被搜索到的全过程，图中把坐标系分成了 4 个象限，第一象限是用户，第二象限是原始文档，第三象限是 Elasticsearch，第四象限是搜索结果。

首先看索引过程。在第二象限中有一条原始的文档，该文档有 title 和 content 两个字段。当把这条文档写入 Elasticsearch 之后，默认情况下 Elasticsearch 中会保存两份内容，一份是该文档的原始内容，也就是_source 中的文档内容，另一份是索引时通过分词、过滤等一系列过程生成的倒排索引文件，倒排索引中保存了词项和文档的对应关系。

再来看搜索过程。第一象限的用户对文档进行搜索，Elasticsearch 接收到查询关键词之后到倒排索引中进行查询，通过倒排索引中维护的倒排记录表找到关键词对应的文档集合，然后做评分、排序、高亮处理，最终返回搜索结果给用户。

搜索机制解决的是相关度的问题，当用户输入一个查询，Elasticsearch 通过排序模型计算文档和查询关键词之间的相关度，按照评分排序后返回最相关的文档给用户。另外，Elasticsearch 中还有一种过滤机制，过滤机制解决的是只根据条件对文档进行过滤，不计算评分。

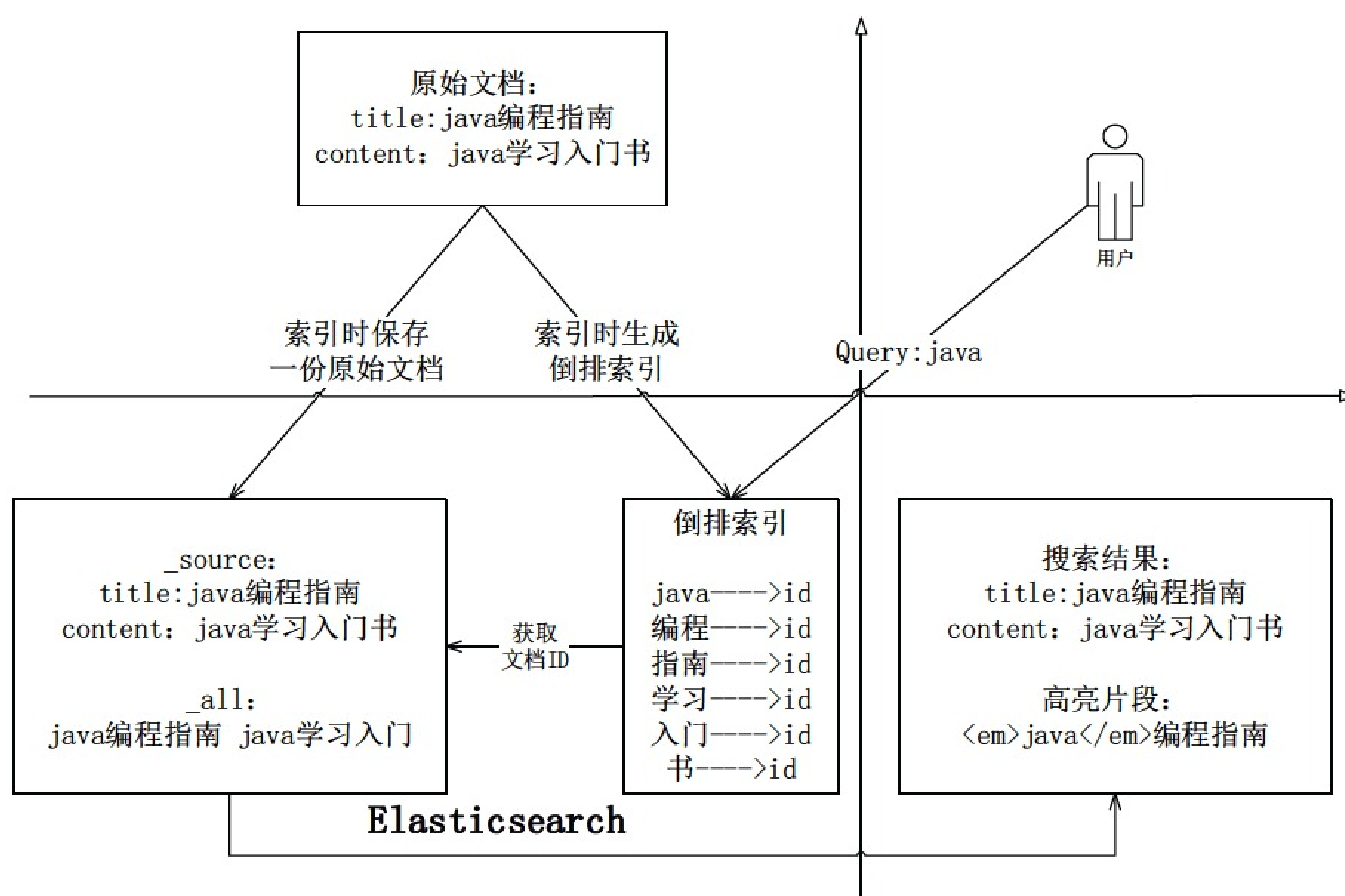


图 6-1 搜索流程图

为了有助于学习搜索的相关知识，这里准备一些 IT 类图书信息作为测试数据，每一条文档都包含 id、标题、编程语言、作者、定价、出版日期和内容描述这 7 个字段。首先把下面的内容保存到 books.json 文件中:

```
{"index":{ "_index": "books", "_type": "IT", "_id": "1" }}
{"id":"1","title":"Java 编程思想","language":"java","author":"Bruce
Eckel", "price":70.20, "publish_time":"2007-10-01","description":
"Java 学习必读经典,殿堂级著作！赢得了全球程序员的广泛赞誉。"}
{"index":{ "_index": "books", "_type": "IT", "_id": "2" }}
{"id":"2","title":"Java 程序性能优化","language":"java","author":"葛一鸣
","price":46.50, "publish_time":"2012-08-01","description":"让你的
Java 程序更快、更稳定。深入剖析软件设计层面、代码层面、JVM 虚拟机层面的优化方法"}
{"index":{ "_index": "books", "_type": "IT", "_id": "3" }}
{"id":"3","title":"Python 科学计算","language":"python","author":"张若
愚","price":81.40, "publish_time":"2016-05-01","description":"零基础学
python,光盘中作者独家整合开发 winPython 运行环境，涵盖了 Python 各个扩展库"}
{"index":{ "_index": "books", "_type": "IT", "_id": "4" }}
{"id":"4","title":"Python 基础教程","language":"python","author":
"Helant","price":54.50, "publish_time":"2014-03-01","description":"
经典的 Python 入门教程，层次鲜明，结构严谨，内容翔实"}
{"index":{ "_index": "books", "_type": "IT", "_id": "5" }}
{"id":"5","title":"JavaScript 高级程序设计","language": "javascript",
"author":"Nicholas C. Zakas","price":66.40,"publish_time":"2012-10-
01","description":"JavaScript 技术经典名著"}
```

创建索引并设置 settings 和 mapping，这里指定索引名为 books，类型名为 IT，副本数为 1，分片数为 3，命令如下：

```
PUT books
{
  "settings": {
    "number_of_replicas": 1,
    "number_of_shards": 3
  },
  "mappings": {
    "IT":{
      "properties": {
        "id":{
          "type": "long"
        },
        "title":{
          "type": "text",
          "analyzer": "ik_max_word"
        },
        "language":{
          "type": "keyword"
        },
        "author":{
          "type": "keyword"
        },
        "price":{
          "type": "double"
        },
        "year":{
          "type": "date",
          "format": "yyy-MM-dd"
        },
        "description":{
          "type": "text",
           "analyzer": "ik_max_word"
        }
      }
    }
  }
}
```

最后执行 bulk 批量导入命令把文档写入 Elasticsearch：

```
curl -XPOST "http://localhost:9200/_bulk?pretty" --data-binary @books.json
```

如果一切顺利，数据导入成功之后就可以搜索了。Elasticsearch RESTful 的查询语句要封装成 JSON 格式的对象，称之为查询 DSL。在 Kibana 的 Dev Tools 中执行 match_all query 查看全部数据：

```
GET books/_search
{
  "query": {
    "match_all": {}
  }
}
```

match_all query 会返回所有文档，文档的得分都是 1。match_all query 可以简写如下：

```
GET books/_search
```

下面以 term query 为例，介绍如何进行词项搜索、分页限制、返回指定字段、显示版本号、控制最小评分以及关键字的高亮。

term 查询用来查找指定字段中包含给定单词的文档，term 查询不被解析，只有查询词和文档中的词精确匹配才会被搜索到，应用场景为查询人名、地名等需要精准配备的需求。比如查询 title 字段中含有关键词“思想”的书籍，查询命令如下：

```
GET  books/_search
{
  "query": {
    "term": { "title": "思想"}
  }
}
```

返回结果如下：

```
{
  "took": 0,
  "timed_out": false,
  "_shards": {
    "total": 3,
    "successful": 3,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "max_score": 0.6099695,
    "hits": [
      {
        "_index": "books",
        "_type": "IT",
        "_id": "1",
```

```
          "_score": 0.6099695,
          "_source": {
            "id": "1",
            "title": "Java 编程思想",
            "language": "java",
            "author": "Bruce Eckel",
            "price": 70.2,
            "publish_time": "2007-10-01",
            "description": "Java 学习必读经典,殿堂级著作！赢得了全球程序员的广泛赞誉。"
          }
        }
      ]
    }
  }
```

hits中的内容就是搜索到的文档，total字段表示一次查询符合查询条件的文档数，通过term查询搜索到了一条文档，文档的 index、type、id、文档评分以及文档内容都可以在查询结果中看到。

测试数据集文档数目较少，在数据量比较大的情况下，一次查询会返回成百上千条数据，这种情况下就需要对查询结果分页。Elasticsearch 提供了结果分页的两个属性：from 和 size，from 指定返回结果的开始位置，默认值为 0，也就是从头开始返回文档，size 指定了一次返回结果所包含的最大文档数量。比如，返回结果中有 1000 条结果，分成 10 页，那么第二页需要返回第 100 到第 200 条文档，可以设置 from 的取值为 100，size 的取值为 100。改进上述 term 查询，加入分页和结果规模控制，查询体如下：

```
GET  books/_search
{
   "from": 0,
   "size": 100,
   "query": {
      "term": {"title": "思想"}
   }
}
```

默认情况下返回结果中包含了文档的所有字段信息，有时候为了简洁，只需要在查询结果中返回某些字段，Elasticsearch 也是允许的。例如，查询标题中包含关键词 java 的书籍，只返回 title 和 price 字段，查询语句如下：

```
GET  books/_search
{
   "_source": ["title","price"],
   "query": {
      "term": {"title": "java"}
   }
}
```

默认情况下返回结果中不包含文档的版本号，如果需要，可以在查询体的中设置 version 属性为 true：

```
GET  books/_search
{
  "version": true,
  "query": {
    "term": {"title": "java"}
  }
}
```

Elasticsearch 提供了基于最小评分的过滤机制，这种机制非常有用。执行一次搜索相关的文档会非常多，有些文档的相关性比较低，通过评分过滤可以筛选出最相关的文档。要想实现这一功能，在查询体中添加 min_score 的最低评分数，只有评分超过这个分数的文档才会被返回，下面的查询会返回 title 中包含关键词 java 且文档评分不低于 0.6 的文档，查询语句如下：

```
GET books/_search
{
  "min_score": 0.6,
  "query": {
    "term": {
      "title": "java"
    }
  }
}
```

最后再演示如何高亮查询关键字，查询语句如下：

```
GET books/_search
{
  "query": {
    "term": {
      "title": "编程"
    }
  },
  "highlight": {
    "fields": {
         "title": {}
    }
  }
}
```

返回结果如图 6-2 所示。



图 6-2　term query 关键字高亮

到此为止，通过 term 查询演示了如何构造查询语句、如何控制查询结果的规模大小、如何返回查询到的文档的版本号、如何只返回文档的部分字段、如何设置最小评分、如何实现关键词高亮。学习 Elasticsearch 搜索命令的简单流程是先构造查询语句，再执行查询，最后查看返回结果，动手敲命令是最好的学习方法，后面的各种查询学习方法类似，我们只需要改变 query 中的查询语句，就可以实现更多种功能的搜索。

6.2　全文查询

高级别的全文搜索通常用于在全文字段（例如：一封邮件的正文）上进行全文搜索，通过全文查询理解被查询字段是如何被索引和分析的，在执行之前将每个字段的分词器（或搜索分词器）应用于查询字符串。

6.2.1　match query

match 查询会解析查询语句，举一个经典的例子：假设我们现在的查询语句为“java 编程”，查询域为 title，使用 term query 进行查询，查询命令及查询结果如图 6-3 所示，可以看到搜索结果为空。但是测试文档集合中有一个文档的 title 为“Java 编程思想”，查询“java 编程”理应返回该文档的，为什么没有返回？

究其原因，term query 查的是词项，“Java 编程思想”经过分词以后会变成“java”“编程”“思想”等多个词项，不存在词项“java 编程”，因此返回结果为空。换成 match query，查询结果如图 6-4 所示。

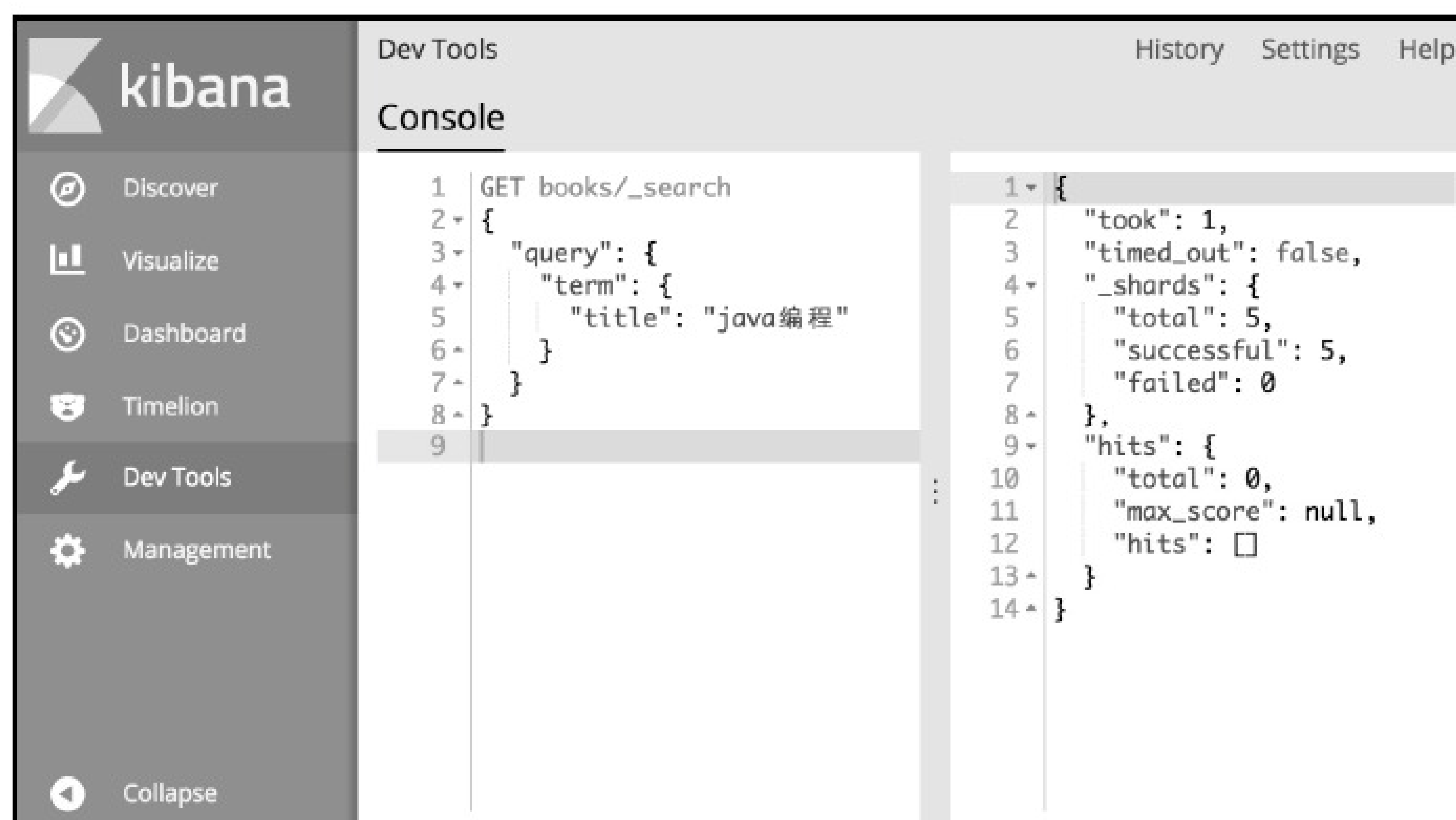


图 6-3　term query 查询结果



图 6-4　match query 查询结果

match query 会对查询语句进行分词，分词后查询语句中的任何一个词项被匹配，文档就会被搜索到。如果想查询匹配所有关键词的文档，可以用 and 操作符连接，命令如下：

```
GET books/_search
{
    "query": {
        "match" : {
            "title" : {
                "query" : "java 编程思想",
                "operator" : "or"
            }
        }
    }
}
```

6.2.2　match_phrase query

match_phrase query 首先会把 query 内容分词，分词器可以自定义，同时文档还要满足以下两个条件才会被搜索到：

（1）分词后所有词项都要出现在该字段中。
（2）字段中的词项顺序要一致。

例如，有以下 3 个文档，使用 match_phrase 查询“hello world”，只有前两个文档会被匹配：

```
PUT test/test/1
{ "foo": "I just said hello world"}

PUT test/test/2
{"foo": "Hello world"}

PUT test/test/3
{"foo": "World Hello"}

GET test/_search
{
  "query": {
    "match_phrase": {"foo": "hello world"}
  }
}
```

6.2.3　match_phrase_prefix query

match_phrase_prefix 和 match_phrase 类似，只不过 match_phrase_prefix 支持最后一个 term 前缀匹配：

```
GET test/_search
{
    "query": {
        "match_phrase_prefix" : {
            "foo" : "hello w"
        }
    }
}
```

6.2.4　multi_match query

multi_match 是 match 的升级，用于搜索多个字段。查询语句为“java 编程”，查询域为 title 和 description，查询语句如下：

```
GET books/_search
{
  "query": {
```

```
      "multi_match" : {
        "query":    "java 编程",
        "fields": [ "title", "description" ]
      }
    }
  }
```

multi_match 支持对要搜索的字段的名称使用通配符，示例如下：

```
GET /_search
{
  "query": {
    "multi_match" : {
      "query":    "java 编程",
      "fields": [ "title", "*_name" ]
    }
  }
}
```

同时，也可以用指数符指定搜索字段的权重。指定关键词出现在 title 中的权重是出现在 description 字段中的 3 倍，命令如下：

```
GET /_search
{
  "query": {
    "multi_match" : {
      "query" : " java 编程",
      "fields" : [ "title^3", " description " ]
    }
  }
}
```

6.2.5　common_terms query

common_terms query 是一种在不牺牲性能的情况下替代停用词提高搜索准确率和召回率的方案。

查询中的每个词项都有一定的代价，以搜索“The brown fox”为例，query 会被解析成三个词项“the”“brown”和“fox”，每个词项都会到索引中执行一次查询。很显然包含“the”的文档非常多，相比其他词项，“the”的重要性会低很多。传统的解决方案是把“the”当作停用词处理，去除停用词之后可以减少索引大小，同时在搜索时减少对停用词的收缩。

虽然停用词对文档评分影响不大，但是当停用词仍然有重要意义的时候，去除停用词就不是完美的解决方案了。如果去除停用词，就无法区分“happy”和“not happy”，“The”“To be or not to be”就不会在索引中存在，搜索的准确率和召回率就会降低。

common_terms query 提供了一种解决方案，它把 query 分词后的词项分成重要词项（低频词项）和不重要的词项（高频词，也就是之前的停用词）。在搜索的时候，首先搜索和重要词

项匹配的文档，这些文档是词项出现较少并且词项对其评分影响较大的文档。然后执行第二次查询，搜索对评分影响较小的高频词项，但是不计算所有文档的评分，而是只计算第一次查询已经匹配的文档得分。如果一个查询中只包含高频词，那么会通过 and 连接符执行一个单独的查询，换言之，会搜索所有的词项。

词项是高频词还是低频词是通过 cutoff_frequency 来设置阀值的，取值可以是绝对频率（频率大于 1）或者相对频率（0～1）。common_terms query 最有趣之处在于它能自适应特定领域的停用词，例如，在视频托管网站上，诸如“clip”或“video”之类的高频词项将自动表现为停用词，无须保留手动列表。

例如，文档频率高于 0.1%的词项将会被当作高频词项，词频之间可以用 low_freq_operator、high_freq_operator 参数连接。设置低频词操作符为“and”使所有的低频词都是必须搜索的，示例代码如下：

```
GET /_search
{
   "query": {
      "common": {
         "body": {
            "query": "nelly the elephant as a cartoon",
             "cutoff_frequency": 0.001,
             "low_freq_operator": "and"
         }
      }
   }
}
```

上述操作等价于：

```
GET /_search
{
   "query": {
      "bool": {
         "must": [
         { "term": { "body": "nelly"}},
         { "term": { "body": "elephant"}},
         { "term": { "body": "cartoon"}}
         ],
         "should": [
         { "term": { "body": "the"}},
         { "term": { "body": "as"}},
         { "term": { "body": "a"}}
         ]
      }
   }
}
```

6.2.6 query_string query

query_string query 是与 Lucene 查询语句的语法结合非常紧密的一种查询，允许在一个查询语句中使用多个特殊条件关键字（如：AND|OR|NOT ）对多个字段进行查询，建议熟悉 Lucene 查询语法的用户去使用。

6.2.7 simple_query_string

simple_query_string 是一种适合直接暴露给用户，并且具有非常完善的查询语法的查询语句，接受 Lucene 查询语法，解析过程中发生错误不会抛出异常。例子如下：

```
GET /_search
{
  "query": {
    "simple_query_string" : {
        "query": "\"fried eggs\" +(eggplant | potato) -frittata",
        "analyzer": "snowball",
        "fields": ["body^5","_all"],
        "default_operator": "and"
    }
  }
}
```

6.3 词项查询

全文搜索在执行查询之前会分析查询字符串，词项搜索时对倒排索引中存储的词项进行精确操作。词项级别的查询通常用于结构化数据，如数字、日期和枚举类型。

6.3.1 term query

term query 用于词项搜索，在 6.1 小节中已经解释过了，这里不再重复。

6.3.2 terms query

terms 查询是 term 查询的升级，可以用来查询文档中包含多个词的文档。比如想查询 title 字段中包含关键词“java”或“python”的文档，构造查询语句如下：

```
GET books/_search
{
  "query": {
    "terms": {
      "title": ["java","python"]
    }
  }
}
```

6.3.3　range query

range 查询用于匹配在某一范围内的数值型、日期类型或者字符串型字段的文档，比如搜索哪些书籍的价格在 50 到 100 之间、哪些书籍的出版时间在 2014 年到 2016 年之间。使用 range 查询只能查询一个字段，不能作用在多个字段上。range 查询支持的参数有以下几种：

- gt　大于，查询范围的最小值，也就是下界，但是不包含临界值。
- gte　大于等于，和 gt 的区别在于包含临界值。
- lt　小于，查询范围的最大值，也就是上界，但是不包含临界值。
- lte　小于等于，和 lt 的区别在于包含临界值。

例如，想要查询价格大于 50 小于等于 70 的书籍，即 50<price<=70，构造查询语句如下：

```
GET books/_search
{
  "query": {
    "range": {
      "price": {
        "gt": 50,
        "lte": 70
      }
    }
  }
}
```

查询出版日期在 2016 年 1 月 1 日和 2016 年 12 月 31 之间的书籍，对 publish_time 字段进行 range 查询，命令如下：

```
GET books/_search
{
    "query": {
        "range" : {
            "publish_time" : {
                "gte": "2016-01-01",
                "lte": "2016-12-31",
                "format": "yyyy-MM-dd"
            }
        }
    }
}
```

6.3.4　exists query

exists 查询会返回字段中至少有一个非空值的文档。举例说明：

```
{
   "query": {
     "exists": {
```

```
          "field": "user"
        }
      }
    }
```

以下文档会匹配上面的查询：

- { "user": "jane" }　有 user 字段，且不为空。
- { "user": "" }　有 user 字段，值为空字符串。
- { "user": "-" }　有 user 字段，值不为空。
- { "user": ["jane"] }　有 user 字段，值不为空。
- { "user": ["jane", null]　有 user 字段，至少一个值不为空即可。

下面的文档都不会被匹配：

- { "user": null }　虽然有 user 字段，但是值为空。
- { "user": [] }　虽然有 user 字段，但是值为空。
- { "user": [null] }　虽然有 user 字段，但是值为空。
- { "foo":　"bar" }　没有 user 字段。

6.3.5　prefix query

prefix 查询用于查询某个字段中以给定前缀开始的文档，比如查询 title 中含有以 java 为前缀的关键词的文档，那么含有 java、javascript、javaee 等所有以 java 开头关键词的文档都会被匹配。查询 description 字段中包含有以 win 为前缀的关键词的文档，查询语句如下：

```
GET books/_search
{
  "query": {
    "prefix": {
      "description": "win"
    }
  }
}
```

6.3.6　wildcard query

wildcard query 中文译为通配符查询，支持单字符通配符和多字符通配符，?用来匹配一个任意字符，*用来匹配零个或者多个字符。以 H?tland 为例，Hatland、Hbtland 等都可以匹配，但是不能匹配 Htland，?只能代表一位。H*tland 可以匹配 Htland、Habctland 等，*可以代表 0 至多个字符。和 prefix 查询一样，wildcard 查询的查询性能也不是很高，需要消耗较多的 CPU 资源。

下面举一个 wildcard 查询的例子，假设需要找某一作者写的书，但是忘记了作者名字的全称，只记住了前两个字，那么就可以使用通配符查询，查询语句如下：

```
GET books/_search
{
   "query": {
     "wildcard": {
```

```
            "author": "张若*"
        }
    }
}
```

6.3.7 regexp query

Elasticsearch 也支持正则表达式查询，通过 regexp query 可以查询指定字段包含与指定正则表达式匹配的文档。可以代表任意字符，“a.c.e”和“ab...”都可以匹配“abcde”，a{3}b{3}、a{2,3}b{2,4}、a{2,}{2,}都可以匹配字符串“aaabbb”。

例如需要匹配以 W 开头紧跟着数字的邮政编码，使用正则表达式查询构造查询语句如下：

```
GET _search
{
    "query": {
        "regexp": {
            "postcode": "W[0-9].+"
        }
    }
}
```

6.3.8 fuzzy query

编辑距离又称 Levenshtein 距离，是指两个字串之间，由一个转成另一个所需的最少编辑操作次数。许可的编辑操作包括将一个字符替换成另一个字符，插入一个字符，删除一个字符。fuzzy 查询就是通过计算词项与文档的编辑距离来得到结果的，但是使用 fuzzy 查询需要消耗的资源比较大，查询效率不高，适用于需要模糊查询的场景。举例如下，用户在输入查询关键词时不小心把“javascript”拼成“javascritp”，在存在拼写错误的情况下使用模糊查询仍然可以搜索到含有“javascript”的文档，查询语句如下：

```
GET books/_search
{
  "query": {
    "fuzzy": {
      "title": "javascritp"
    }
  }
}
```

6.3.9 type query

type query 用于查询具有指定类型的文档。例如查询 Elasticsearch 中 type 为 IT 的文档，查询语句如下：

```
GET _search
{
    "query": {
        "type" : {
```

```
            "value" : "IT"
         }
      }
   }
```

6.3.10　ids query

ids query 用于查询具有指定 id 的文档。类型是可选的，也可以省略，也可以接受一个数组。如果未指定任何类型，则会搜索索引中的所有类型。例如，查询类型为 IT，id 为 1、3、5 的文档，查询语句如下：

```
GET books/_search
{
  "query": {
    "ids": {
      "type": "IT",
      "values": ["1","3","5" ]
    }
  }
}
```

6.4　复合查询

复合查询就是把一些简单查询组合在一起实现更复杂的查询需求，除此之外复合查询还可以控制另外一个查询的行为。

6.4.1　constant_score query

constant_score query 可以包装一个其他类型的查询，并返回匹配过滤器中的查询条件且具有相同评分的文档。下面的查询语句会返回 title 字段中含有关键词“java”的文档，所有文档的评分都是 1.2：

```
GET books/_search
{
    "query": {
        "constant_score" : {
            "filter" : {
                "term" : { "title" : "java"}
            },
            "boost" : 1.2
        }
    }
}
```

6.4.2 bool query

bool 查询可以把任意多个简单查询组合在一起，使用 must、should、must_not、filter 选项来表示简单查询之间的逻辑，每个选项都可以出现 0 次到多次，它们的含义如下：

- must　文档必须匹配 must 选项下的查询条件，相当于逻辑运算的 AND。
- should　文档可以匹配 should 选项下的查询条件也可以不匹配，相当于逻辑运算的 OR。
- must_not　与 must 相反，匹配该选项下的查询条件的文档不会被返回。
- filter　和 must 一样，匹配 filter 选项下的查询条件的文档才会被返回，但是 filter 不评分，只起到过滤功能。

假设要查询 title 中包含关键词 java，并且 price 不能高于 70，description 可以包含也可以不包含虚拟机的书籍，构造 bool 查询语句如下：

```
GET  books/_search
{
  "query": {
    "bool": {
      "minimum_should_match": 1,
      "must": {
        "match": { "title": "java"}
      },
      "should": [
        {"match": { "description": " 虚拟机"}}
      ],
      "must_not": {
        "range": {"price": {"gte": 70}}
      }
    }
  }
}
```

6.4.3 dis_max query

dis_max query 与 bool query 有一定联系也有一定区别，dis_max query 支持多并发查询，可返回与任意查询条件子句匹配的任何文档类型。与 bool 查询可以将所有匹配查询的分数相结合使的方式不同，dis_max 查询只使用最佳匹配查询条件的分数。请看下面的例子：

```
GET /_search
{
    "query": {
        "dis_max" : {
            "tie_breaker" : 0.7,
            "boost" : 1.2,
            "queries" : [
```

```
                {
                    "term" : { "age" : 34 }
                },
                {
                    "term" : { "age" : 35 }
                }
            ]
        }
    }
}
```

6.4.4 function_score query

function_score query 可以修改查询的文档得分，这个查询在有些情况下非常有用，比如通过评分函数计算文档得分代价较高，可以改用过滤器加自定义评分函数的方式来取代传统的评分方式。

使用 function_score query，用户需要定义一个查询和一至多个评分函数，评分函数会对查询到的每个文档分别计算得分。

下面这条查询语句会返回 books 索引中的所有文档，文档的最大得分为 5，每个文档的得分随机生成，权重的计算模式为相乘模式。

```
GET books/_search
{
    "query": {
        "function_score": {
            "query": { "match_all": {} },
            "boost": "5",
            "random_score": {},
            "boost_mode":"multiply"
        }
    }
}
```

使用脚本自定义评分公式，这里把 price 值的十分之一开方作为每个文档的得分，查询语句如下：

```
GET books/_search
{
    "query": {
        "function_score": {
            "query": {
                "match": { "title": "java" }
            },
            "script_score" : {
                "script" : {
```

```
                    "inline": "Math.sqrt(doc['price'].value/10)"
                  }
              }
          }
      }
  }
```

6.4.5　boosting query

boosting 查询用于需要对两个查询的评分进行调整的场景，boosting 查询会把两个查询封装在一起并降低其中一个查询的评分。下面通过对比来说明 boosting 查询的用法，首先查询 books 索引下 title 字段中含有关键词 python 的文档，结果如图 6-5 所示，可以看到 2014 年出版的《python 基础教程》这本书的评分为 1.0131714，2016 年出版的《python 科学计算》评分为 0.6099695。

图 6-5　不使用 boosting 查询结果

boosting 查询包括 positive、negative 和 negative_boost 三个部分，positive 中的查询评分保持不变，negative 中的查询会降低文档评分，negative_boost 指明 negative 中降低的权值。如果我们想对 2015 年之前出版的书降低评分，可以构造一个 boosting 查询，查询语句如下。

```
GET books/_search
{
  "query": {
    "boosting": {
      "positive": {
        "match": {
          "title": "python"
        }
      },
```

```
        "negative": {
          "range": {
            "publish_time": {
              "lte": "2015-01-01"
            }
          }
        },
        "negative_boost": 0.2
      }
    }
  }
```

Boosting 查询中指定了抑制因子为 0.2，publish_time 的值在 2015-01-01 之后的文档得分不变，publish_time 的值在 2015-01-01 之前的文档得分为原得分的 0.2 倍，查询结果如图 6-6 所示。



图 6-6　使用 boosting 查询结果

6.4.6　indices query

indices query 适用于需要在多个索引之间进行查询的场景，它允许指定一个索引名字列表和内部查询。indices query 中有 query 和 no_match_query 两部分，query 中用于搜索指定索引列表中的文档，no_match_query 中的查询条件用于搜索指定索引列表之外的文档。下面的查询语句实现了搜索索引 books、books2 中 title 字段包含关键字 javascript，其他索引中 title 字段包含 basketball 的文档，查询语句如下：

```
GET _search
{
  "query": {
    "indices": {
```

```
      "indices": [  "books","books2"],
      "query": {"match": {"title": "javascript"}},
      "no_match_query": {"term": {"title": "basketball"}}
    }
  }
}
```

6.5　嵌套查询

在 Elasticsearch 这样的分布式系统中执行全 SQL 风格的连接查询代价昂贵，是不可行的。相应地，为了实现水平规模地扩展，Elasticsearch 提供了以下两种形式的 join：

- nested query（嵌套查询）
  文档中可能包含嵌套类型的字段，这些字段用来索引一些数组对象，每个对象都可以作为一条独立的文档被查询出来。
- has_child query（有子查询）和 has_parent query（有父查询）
  父子关系可以存在单个的索引的两个类型的文档之间。has_child 查询将返回其子文档能满足特定查询的父文档，而 has_parent 则返回其父文档能满足特定查询的子文档。

6.5.1　nested query

文档中可能包含嵌套类型的字段，这些字段用来索引一些数组对象，每个对象都可以作为一条独立的文档被查询出来（用嵌套查询）。

```
PUT /my_index
{
    "mappings": {
        "type1" : {
            "properties" : {
                "obj1" : {
                    "type" : "nested"
                }
            }
        }
    }
}
```

6.5.2　has_child query

文档的父子关系创建索引时在映射中声明，这里以员工（employee）和工作城市（branch）为例，它们属于不同的类型，相当于数据库中的两张表，如果想把员工和他们工作的城市关联起来，需要告诉 Elasticsearch 文档之间的父子关系，这里 employee 是 child type，branch 是 parent type，在映射中声明，执行命令：

```
PUT /company
{
  "mappings": {
    "branch": {},
    "employee": {"_parent": { "type": "branch"}}
  }
}
```

使用 bulk api 索引 branch 类型下的文档，命令如下：

```
POST company/branch/_bulk
{ "index": { "_id": "london" }}
{ "name": "London Westminster", "city": "London", "country": "UK" }
{ "index": { "_id": "liverpool" }}
{ "name": "Liverpool Central", "city": "Liverpool", "country": "UK" }
{ "index": { "_id": "paris" }}
{ "name": "Champs Élysées", "city": "Paris", "country": "France" }
```

添加员工数据：

```
POST company/employee/_bulk
{ "index": { "_id": 1, "parent": "london" }}
{ "name": "Alice Smith", "dob": "1970-10-24", "hobby": "hiking" }
{ "index": { "_id": 2, "parent": "london" }}
{ "name": "Mark Thomas", "dob": "1982-05-16", "hobby": "diving" }
{ "index": { "_id": 3, "parent": "liverpool" }}
{ "name": "Barry Smith", "dob": "1979-04-01", "hobby": "hiking" }
{ "index": { "_id": 4, "parent": "paris" }}
{ "name": "Adrien Grand", "dob": "1987-05-11", "hobby": "horses" }
```

通过子文档查询父文档要使用 has_child 查询。例如，搜索 1980 年以后出生的员工所在的分支机构，employee 中 1980 年以后出生的有 Mark Thomas 和 Adrien Grand，他们分别在 london 和 paris，执行以下查询命令进行验证：

```
GET company/branch/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "query": {
        "range": {"dob": {"gte": "1980-01-01"}}
      }
    }
  }
}
```

搜索哪些机构中有名为“Alice Smith”的员工，因为使用 match 查询，会解析为“Alice”和“Smith”，所以 Alice Smith 和 Barry Smith 所在的机构会被匹配，执行以下查询命令进行验证：

```
GET /company/branch/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "score_mode": "max",
      "query": {"match": { "name": "Alice Smith"}}
    }
  }
}
```

可以使用 min_children 指定子文档的最小个数。例如，搜索最少含有两个 employee 的机构，查询命令如下：

```
GET /company/branch/_search?pretty
{
  "query": {
    "has_child": {
      "type": "employee",
      "min_children": 2,
      "query": {"match_all": {}}
    }
  }
}
```

6.5.3　has_parent query

通过父文档查询子文档使用 has_parent 查询。比如，搜索哪些 employee 工作在 UK，查询命令如下：

```
GET /company/employee/_search
{
  "query": {
    "has_parent": {
      "parent_type": "branch",
      "query": {"match": {"country": "UK"}}
    }
  }
}
```

6.6　位置查询

Elasticsearch 可以对地理位置点 geo_point 类型和地理位置形状 geo_shape 类型的数据进行搜索。为了学习方便，这里准备一些城市的地理坐标作为测试数据，每一条文档都包含城市名称和地理坐标这两个字段，这里的坐标点取的是各个城市中心的一个位置。首先把下面的内容保存到 geo.json 文件中：

```
{"index":{ "_index": "geo", "_type": "city", "_id": "1" }}
{"name":"北京","location":"39.9088145109,116.3973999023"}
{"index":{ "_index": "geo", "_type": "city", "_id": "2" }}
{"name":"乌鲁木齐","location":"43.8266300000,87.6168800000"}
{"index":{ "_index": "geo", "_type": "city", "_id": "3" }}
{"name":"西安","location":"34.3412700000,108.9398400000"}
{"index":{ "_index": "geo", "_type": "city", "_id": "4" }}
{"name":"郑州","location":"34.7447157466,113.6587142944"}
{"index":{ "_index": "geo", "_type": "city", "_id": "5" }}
{"name":"杭州","location":"30.2294080260,120.1492309570"}
{"index":{ "_index": "geo", "_type": "city", "_id": "6" }}
{"name":"济南","location":"36.6518400000,117.1200900000"}
```

创建一个索引并设置映射：

```
PUT geo
{
  "mappings": {
    "city":{
      "properties": {
        "name":{
          "type": "keyword"
        },
        "location":{
          "type": "geo_point"
        }
      }
    }
  }
}
```

然后执行批量导入命令：

```
curl -XPOST "http://localhost:9200/_bulk?pretty" --data-binary @geo.json
```

6.6.1　geo_distance query

geo_distance query 可以查找在一个中心点指定范围内的地理点文档。例如，查找距离天津 200km 以内的城市，搜索结果中会返回北京，命令如下：

```
GET geo/_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_distance" : {
                    "distance" : "200km",
                    "location" : {
                        "lat" : 39.0851000000,
                        "lon" : 117.1993700000
                    }
                }
            }
        }
    }
}
```

按各城市离北京的距离排序：

```
GET geo/_search
{
  "query": {
    "match_all": {}
  },
  "sort": [
    {
      "_geo_distance": {
        "location": "39.9088145109,116.3973999023",
        "unit": "km"
      }
    }
  ]
}
```

6.6.2　geo_bounding_box query

geo_bounding_box query 用于查找落入指定的矩形内的地理坐标。查询中由两个点确定一

个矩形，例如图 6-7 中的银川和南昌，在这两个点上分别做垂线（经度）和平行线（纬度），相交线会组成一个矩形区域。执行下面的查询，可以查询到西安和郑州这两个城市。



图 6-7　银川和南昌确定的矩形

```
GET geo/_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_bounding_box" : {
                    "location" : {
                        "top_left" : {
                            "lat" :38.4864400000,
                            "lon" :106.2324800000
                        },
                        "bottom_right" : {
                            "lat" :28.6820200000,
                            "lon" : 115.8579400000
                        }
                    }
                }
            }
        }
    }
}
```

6.6.3　geo_polygon query

geo_polygon query 用于查找在指定多边形内的地理点。例如，呼和浩特、重庆、上海三地组成一个三角形（见图 6-8），查询位置在该三角形区域内的城市，命令如下：

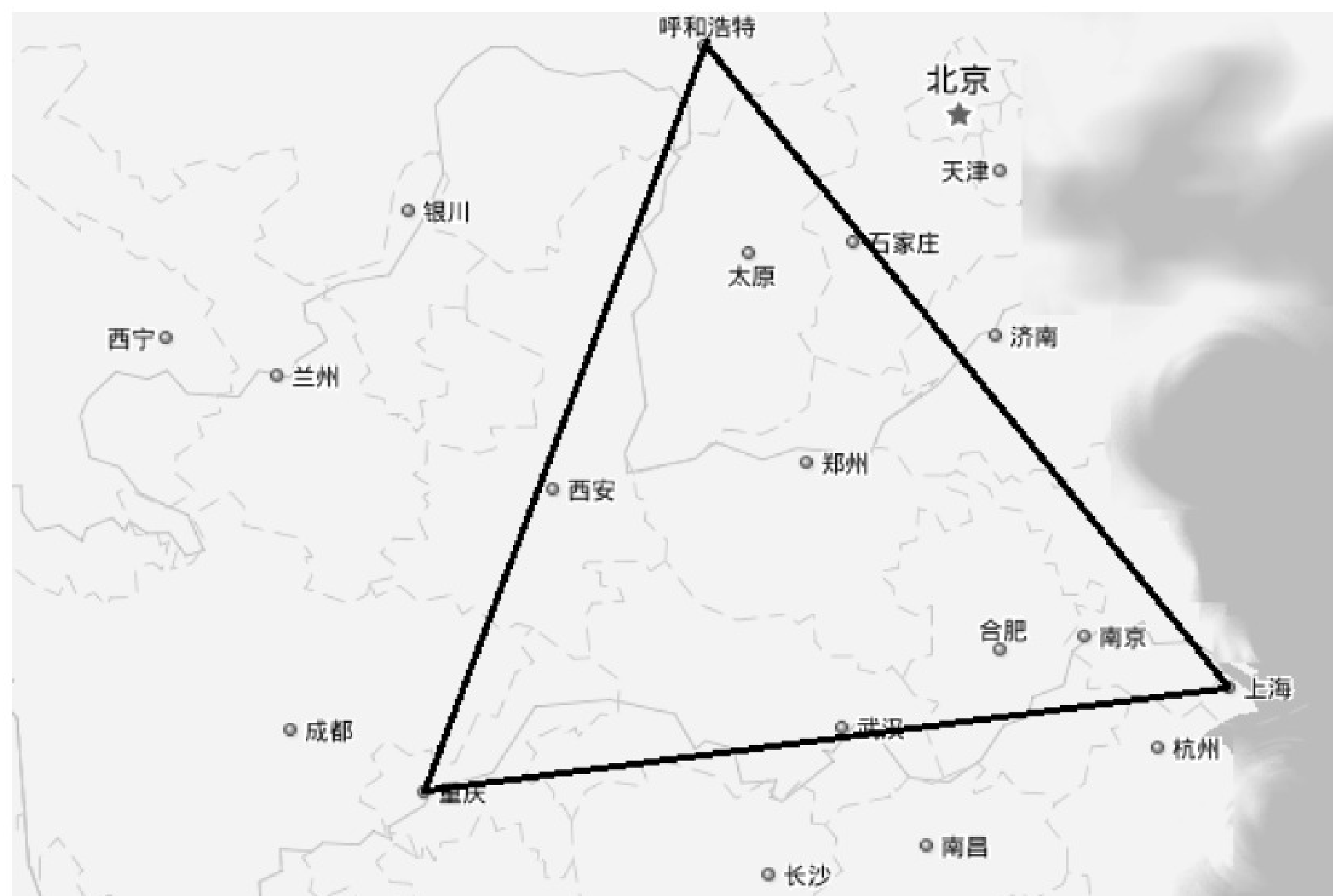


图 6-8　呼和浩特、重庆和上海确定的矩形

```
GET geo/_search
{
    "query": {
        "bool" : {
            "must" : {
                "match_all" : {}
            },
            "filter" : {
                "geo_polygon" : {
                    "location" : {
                        "points" : [
                        {"lat" : 40.8414900000, "lon" : 111.7519900000},
                        {"lat" : 29.5647100000, "lon" : 106.5507300000},
                        {"lat" : 31.2303700000, "lon" : 121.4737000000}
                        ]
                    }
                }
            }
        }
    }
}
```

6.6.4 geo_shape query

geo_shape query 用于查询 geo_shape 类型的地理数据，地理形状之间的关系有相交、包含、不相交三种。创建一个新的索引用于测试，其中 location 字段的类型设为 geo_shape 类型：

```
PUT geoshape
{
  "mappings": {
    "city": {
      "properties": {
        "name": {
          "type": "keyword"
        },
        "location": {
          "type": "geo_shape"
        }
      }
    }
  }
}
```

关于经纬度的顺序这里做一个说明，geo_point 类型的字段纬度在前经度在后，但是对于 geo_shape 类型中的点，是经度在前纬度在后，这一点需要特别注意。

把西安和郑州连成的线写入索引：

```
POST geoshape/city/1
{
  "name":"西安-郑州",
    "location" : {
        "type" : "linestring",
        "coordinates" : [[108.9398400000,34.3412700000],
                         [113.6587142944,34.7447157466]]
    }
}
```

查询包含在由银川和南昌作为对角线上的点组成的矩形的地理形状，由于西安和郑州组成的直线落在该矩形区域内，因此可以被查询到。命令如下：

```
GET geoshape/_search
{
  "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
```

```
      "filter": {
        "geo_shape": {
          "location": {
            "shape": {
              "type": "envelope",
              "coordinates": [
                [106.23248, 38.48644],
                [115.85794, 28.68202]
              ]
            },
            "relation": "within"
          }
        }
      }
    }
  }
}
```

6.7　特殊查询

6.7.1　more_like_this query

more_like_this query 可以查询和提供文本类似的文档，通常用于近似文本的推荐等场景。查询命令如下：

```
GET books/_search
{
    "query": {
        "more_like_this" : {
            "fields" : ["title", "description"],
            "like" : "java virtual machine",
            "min_term_freq" : 1,
            "max_query_terms" : 12
        }
    }
}
```

可选的参数及取值说明如下：

- fields　要匹配的字段，默认是_all 字段。
- like　要匹配的文本。
- min_term_freq　文档中词项的最低频率，默认是 2，低于此频率的文档会被忽略。
- max_query_terms　query 中能包含的最大词项数目，默认为 25。

- min_doc_freq　最小的文档频率，默认为 5。
- max_doc_freq　最大文档频率。
- min_word_length　单词的最小长度。
- max_word_length　单词的最大长度。
- stop_words　停用词列表。
- analyzer　分词器。
- minimum_should_match　文档应匹配的最小词项数，默认为 query 分词后词项数的 30%。
- boost_terms　词项的权重。
- include　是否把输入文档作为结果返回。
- boost　整个 query 的权重，默认为 1.0。

6.7.2　script query

Elasticsearch 支持使用脚本进行查询。例如，查询价格大于 80 的文档，命令如下：

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "script": {
            "script": {
              "inline": "doc['price'].value > 80",
              "lang": "painless"
            }
          }
        }
      ]
    }
  }
}
```

6.7.3　percolate query

一般情况下，我们是先把文档写入到 Elasticsearch 中，通过查询语句对文档进行搜索。percolate query 则是反其道而行之的做法，它会先注册查询条件，根据文档来查询 query。例如，在 my-index 索引中有一个 laptop 类型，文档有 price 和 name 两个字段，在映射中声明一个 percolator 类型的 query，命令如下：

```
PUT my-index
{
    "mappings": {
        "laptop": {
```

```
            "properties": {
                "price": {"type": "long"},
                "name":{"type": "text"}
            }
        },
        "queries": {
            "properties": {
                "query": {"type": "percolator"}
            }
        }
    }
}
```

注册一个 bool query，bool query 中包含一个 range query，要求 price 字段的取值小于等于 10000，并且 name 字段中含有关键词 macbook：

```
PUT /my-index/queries/1?refresh
{
  "query": {
    "bool": {
      "must": [
       {"range": {"price": {"lte": 10000}}},
       { "match": {"name": "macbook" } }
      ]
    }
  }
}
```

通过文档查询 query：

```
GET /my-index/_search
{
    "query" : {
        "percolate" : {
            "field" : "query",
            "document_type" : "laptop",
            "document" : {
                "price" : 9999,
                "name":"macbook pro on sale"
            }
        }
    }
}
```

文档符合 query 中的条件，返回结果中可以查到上文中注册的 bool query。percolate query 的这种特性适用于数据分类、数据路由、事件监控和预警等场景。

6.8　搜索高亮

6.8.1　自定义高亮片段

在前面的基本查询中，我们简单地使用了高亮功能标记查询关键字，Elasticsearch 默认会用标签标记关键字。如果我们想使用自定义标签，在高亮属性中给需要高亮的字段加上 pre_tags 和 post_tags 即可。例如，搜索 title 字段中包含关键词 javascript 的书籍并使用自定义 HTML 标签高亮关键词，查询语句如下：

```
GET books/_search
{
  "query": {
    "match": { "title": "javascript"}
  },
  "highlight": {
    "fields": {
      "title": {
        "pre_tags": ["<strong>"],
        "post_tags": ["</strong>"]
      }
    }
  }
}
```

6.8.2　多字段高亮

关于搜索高亮，还需要掌握如何设置多字段搜索高亮。比如，搜索 title 字段的时候，我们期望 description 字段中的关键字也可以高亮，这时候就需要把 require_field_match 属性的取值设置为 fasle。require_field_match 的默认值为 true，只会高亮匹配的字段。多字段高亮的查询语句如下：

```
GET books/_search
{
   "query": {
      "match": {
         "title": "javascript"
      }
   },
   "highlight": {
      "require_field_match": false,
      "fields": {
         "title": {},
```

```
          "description": {}
        }
      }
    }
```

6.8.3　高亮性能分析

Elasticsearch 提供了三种高亮器，分别是默认的 highlighter 高亮器、postings-highlighter 高亮器和 fast-vector-highlighter 高亮器。

默认的 highlighter 是最基本的高亮器。highlighter 高亮器实现高亮功能需要对_source 中保存的原始文档进行二次分析，其速度在三种高亮器里最慢，优点是不需要额外的存储空间。

postings-highighlighter 高亮器实现高亮功能不需要二次分析，但是需要在字段的映射中设置 index_options 参数的取值为 offsets，即保存关键词的偏移量，速度快于默认的 highlighter 高亮器。例如，配置 comment 字段使用 postings-highighlighter 高亮器，映射如下：

```
PUT /example
{
  "mappings": {
    "doc" : {
      "properties": {
        "comment" : {
          "type": "text",
          "index_options" : "offsets"
        }
      }
    }
  }
}
```

fast-vector-highlighter 高亮器实现高亮功能速度最快，但是需要在字段的映射中设置 term_vector 参数的取值为 with_positions_offsets，即保存关键词的位置和偏移信息，占用的存储空间最大，是典型的空间换时间的做法。例如，配置 comment 字段使用 fast-vector-highlighter 高亮器，映射如下：

```
PUT /example
{
  "mappings": {
    "doc" : {
      "properties": {
        "comment" : {
          "type": "text",
          "term_vector" : "with_positions_offsets"
        }
      }
    }
  }
}
```

6.9　搜索排序

6.9.1　默认排序

Elasticsearch 是按照查询和文档的相关度进行排序的，默认按评分降序排序，搜索 title 字段中包含 java 关键词的文档：

```
GET  books/_search
{
  "query": {
    "term": { "title": "java"}
  }
}
```

等价于：

```
GET books/_search
{
  "query": {
    "term": {"title": "java"}
  },
  "sort": [
    {"_score": {"order": "desc"}}
  ]
}
```

对于 match_all query 而言，由于只返回所有文档，不需要评分，文档的顺序为添加文档的顺序。如果需要改变 match_all query 的文档返回顺序，可以对_doc 进行排序。例如，返回最后添加的那条文档，可以对_doc 降序排序，设置返回文档条数为 1，命令如下：

```
GET books/_search
{
  "size":1,
  "query": {
    "match_all": {}
  },
  "sort": [{"_doc": { "order": "desc" }}
  ]
}
```

6.9.2　多字段排序

和 SQL 类型，Elasticsearch 也支持多字段排序。例如，先按价格降序排序，价格相等的按照出版年份升序排序，命令如下：

```
GET books/_search
{
  "sort": [
    {"price": { "order": "desc"} },
    {"year": {"order": "asc"}}
  ]
}
```

6.9.3　分片影响评分

Elasticsearch 5.4 之后对于 text 类型的字段，默认采用的是 BM25 评分模型，而不是基于 tf-idf 的向量空间模型，评分模型的选择可以通过 similarity 参数在映射中指定。需要注意的是，Elasticsearch 是在每一个分片上单独打分的，分片的数量会影响打分的结果。

下面进行分片对评分影响的测试，products 索引 product 类型下的 name 字段，数据类型为 text 文本类型，分词器为 ik_smart，评分模型为 tf-idf，索引的分片数为 1，执行以下创建索引的命令：

```
PUT products
{
  "settings": {
    "number_of_shards": 1
  },
  "mappings": {
    "product": {
      "properties": {
        "name": {
          "type": "text",
          "analyzer": "ik_smart",
          "similarity": "classic"
        }
      }
    }
  }
}
```

写入三条测试文档：

```
PUT products/product/1
{"name":"柠檬"}

PUT products/product/2
{"name":"柠檬汽水"}

PUT products/product/3
{"name":"饮料"}
```

match query 搜索关键词“柠檬”：

```
POST products/_search
{
  "query": {
    "match": {
      "name": "柠檬"
    }
  }
}
```

评分结果：

```
{
  "hits": {
    "total": 2,
    "max_score": 1.658125,
    "hits": [
      {
        "_index": "products",
        "_type": "product",
        "_id": "1",
        "_score": 1.658125,
        "_source": {
          "name": "柠檬"
        }
      },
      {
        "_index": "products",
        "_type": "product",
        "_id": "2",
        "_score": 1.0363282,
        "_source": {
          "name": "柠檬汽水"
        }
      }
    ]
  }
}
```

删除索引，分片数改为 3，重复上述操作，评分结果：

```
{
  "hits": {
    "total": 2,
    "max_score": 1.9753323,
```

```
      "hits": [
        {
          "_index": "products",
          "_type": "product",
          "_id": "1",
          "_score": 1.9753323,
          "_source": {
            "name": "柠檬"
          }
        },
        {
          "_index": "products",
          "_type": "product",
          "_id": "2",
          "_score": 0.625,
          "_source": {
            "name": "柠檬汽水"
          }
        }
      ]
    }
  }
```

对比就可以发现，分片数的改变会影响文档的评分结果，原因就是打分是在每个分片上单独进行的。同时，分词器也会影响评分，原因是使用不同的分词器会使倒排索引中的词项数发生改变，最终影响评分。

6.10　本章小结

全文搜索是 Elasticsearch 的强项，本章首先演示了 Elasticsearch 的搜索机制，之后介绍了 Elasticsearch 全文查询、词项查询、复合查询、嵌套查询、位置查询、特殊查询以及搜索高亮和搜索排序。

第 7 章

聚合分析

众所周知，Elasticsearch 是一个分布式的全文搜索引擎，索引和搜索是 Elasticsearch 的基本功能。事实上，Elasticsearch 的聚合（Aggregations）功能也十分强大，允许在数据上做复杂的分析统计。Elasticsearch 提供的聚合分析功能主要有指标聚合、桶聚合、管道聚合和矩阵聚合四大类，管道聚合和矩阵聚合官方说明是在试验阶段，后期会完全更改或者移除，这里不再对管道聚合和矩阵聚合进行讲解。下面仍以 books 索引中的数据为例，介绍在 Elasticsearch 中进行聚合分析。

7.1 指标聚合

7.1.1 Max Aggregation

Max Aggregation 用于最大值统计。例如，统计 books 索引中价格最高的是哪本书，查询语句如下：

```
GET books/_search
{
  "size": 0,
  "aggs": {
    "max_price": {
      "max": {"field": "price"}
    }
  }
}
```

聚合结果如下：

```
{
  "aggregations": {
```

```
            "max_price": {
                "value": 81.4
            }
        }
    }
```

7.1.2　Min Aggregation

Min Aggregation 用于最小值统计。例如，统计 books 索引中最早出版的是哪本书，查询语句如下：

```
GET books/_search
{
  "size": 0,
  "aggs": {
    "min_year": {
      "min": {
        "field": "publish_time"
      }
    }
  }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "min_year": {
      "value": 1191196800000,
      "value_as_string": "2007-10-01T00:00:00.000Z"
    }
  }
}
```

7.1.3　Avg Aggregation

Avg Aggregation 用于计算平均值。例如，计算 books 索引中所有书的平均价格，查询语句如下：

```
GET books/_search
{
  "size": 0,
  "aggs": {
    "avg_price": {
      "avg": {"field": "price"}
    }
  }
}
```

聚合结果如下：

```
{
    "aggregations": {
        "avg_price": { "value": 63.8}
    }
}
```

7.1.4 Sum Aggregation

Sum Aggregation 用于计算总和。例如，计算 books 索引中所有书的总价，查询语句如下：

```
GET books/_search
{
   "size": 0,
   "aggs": {
      "sum_price": {
         "sum": {"field": "price"}
      }
   }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "sum_price": {
      "value": 319
    }
  }
}
```

7.1.5 Cardinality Aggregation

Cardinality Aggregation 用于基数统计，其作用是先执行类似 SQL 中的 distinct 操作，去掉集合中的重复项，然后统计排重后的集合长度。例如，在 books 索引中对 language 字段进行 cardinality 操作可以统计出编程语言的种类数，查询语句如下：

```
GET books/_search
{
   "size": 0,
   "aggs": {
      "all_lan": {
         "cardinality": {"field": "language"}
      }
   }
}
```

聚合结果如下：

```
{
    "aggregations": {
        "all_lan": {"value": 3}
    }
}
```

7.1.6　Stats Aggregation

Stats Aggregation 用于基本统计，会一次返回 count、max、min、avg 和 sum 这 5 个指标。例如，在 books 索引中对 price 字段进行基本统计，查询语句如下：

```
GET books/_search
{
    "size": 0,
    "aggs": {
        "grades_stats": {
            "stats": {"field": "price"}
        }
    }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "grades_stats": {
      "count": 5,
      "min": 46.5,
      "max": 81.4,
      "avg": 63.8,
      "sum": 319
    }
  }
}
```

7.1.7　Extended Stats Aggregation

Extended Stats Aggregation 用于高级统计，和基本统计功能类似，但是会比基本统计多 4 个统计结果：平方和、方差、标准差、平均值加/减两个标准差的区间。对 books 索引中的 price 字段进行高级统计，查询语句如下：

```
GET books/_search
{
    "size": 0,
    "aggs": {
```

```
        "grades_stats": {
            "extended_stats": {"field": "price"}
        }
    }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "grades_stats": {
      "count": 5,
      "min": 46.5,
      "max": 81.4,
      "avg": 63.8,
      "sum": 319,
      "sum_of_squares": 21095.46,
      "variance": 148.65199999999967,
      "std_deviation": 12.19229264740638,
      "std_deviation_bounds": {
        "upper": 88.18458529481276,
        "lower": 39.41541470518724
      }
    }
  }
}
```

7.1.8　Percentiles Aggregation

Percentiles Aggregation 用于百分位统计。百分位数是一个统计学术语，如果将一组数据从大到小排序，并计算相应的累计百分位，某一百分位所对应数据的值就称为这一百分位的百分位数。例如，对 books 索引中的 price 字段进行百分位统计，查询语句如下：

```
GET books/_search
{
    "size": 0,
    "aggs": {
        "book_price": {
            "percentiles": {"field": "price"}
        }
    }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "book_price": {
```

```
        "values": {
          "1.0": 46.82,
          "5.0": 48.1,
          "25.0": 54.5,
          "50.0": 66.4,
          "75.0": 70.2,
          "95.0": 79.16,
          "99.0": 80.95200000000001
        }
      }
    }
  }
```

7.1.9 Value Count Aggregation

Value Count Aggregation 可按字段统计文档数量。例如，统计 books 索引中包含 author 字段的文档数量，查询语句如下：

```
POST books/_search
{
  "size": 0,
  "aggs": {
    "doc_count": {
      "value_count": {
        "field": "author"
      }
    }
  }
}
```

聚合结果如下:

```
{
  "aggregations": {
    "doc_count": {
      "value": 5
    }
  }
}
```

7.2 桶 聚 合

Bucket 可以理解为一个桶，它会遍历文档中的内容，凡是符合某一要求的就放入一个桶中，分桶相当于 SQL 中的 group by。以 books 索引中的图书为例，一本书会被划分到科技类、经济类

或者其他分类中，那么科技类图书就是一个桶，经济类图书也是一个桶，桶就是符合某一划分标准的文档集合。

7.2.1　Terms Aggregation

Terms Aggregation 用于分组聚合。例如，根据 language 字段对 books 索引中的文档进行分组，统计属于各编程语言的书的数量，构造查询语句如下：

```
POST books/_search?size=0
{
    "aggs": {
        "per_count": {
            "terms": {
                "field": "language"
            }
        }
    }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "per_count": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "java",
          "doc_count": 2
        },
        {
          "key": "python",
          "doc_count": 2
        },
        {
          "key": "javascript",
          "doc_count": 1
        }
      ]
    }
  }
}
```

在 terms 分桶的基础上，还可以对每个桶进行指标聚合。例如，想统计每一类图书的平均价格，可以先按照 language 字段进行 Terms Aggregation，再进行 Avg Aggregation，查询语句如下：

```
POST books/_search?size=0
{
   "aggs": {
      "per_count": {
         "terms": {"field": "language"},
         "aggs": {
            "avg_price": {
               "avg": {"field": "price"}
            }
         }
      }
   }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "per_count": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "java",
          "doc_count": 2,
          "avg_price": {
            "value": 58.35
          }
        },
        {
          "key": "python",
          "doc_count": 2,
          "avg_price": {
            "value": 67.95
          }
        },
        {
          "key": "javascript",
          "doc_count": 1,
          "avg_price": {
            "value": 66.4
          }
        }
      ]
    }
  }
}
```

7.2.2　Filter Aggregation

Filter Aggregation 是过滤器聚合，可以把符合过滤器中的条件的文档分到一个桶中。例如，计算 title 字段中含有关键词 Java 的文档的平均值，查询语句如下：

```
POST books/_search?size=0
{
    "aggs" : {
        "java_avg_price" : {
            "filter" : { "term": { "title": "java" } },
            "aggs" : {
                "avg_price" : { "avg" : { "field" : "price" } }
            }
        }
    }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "java_avg_price": {
      "doc_count": 2,
      "avg_price": {
        "value": 58.35
      }
    }
  }
}
```

7.2.3　Filters Aggregation

Filters Aggregation 是多过滤器聚合，可以把符合多个过滤条件的文档分到不同的桶中。下面命令中 filters 中包含两个 match query，对每个 query 的查询结果进行分组统计，查询语句如下：

```
POST books/_search?size=0
{
  "aggs": {
    "per_avg_price": {
      "filters": {
        "filters": [
          {"match": {"title": "java"} },
          { "match": {"title": "python"}}
        ]
```

```
      },
      "aggs": {
        "avg_price": {"avg": {"field": "price"}}
      }
    }
  }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "per_avg_price": {
      "buckets": [
        {
          "doc_count": 2,
          "avg_price": {
            "value": 58.35
          }
        },
        {
          "doc_count": 2,
          "avg_price": {
            "value": 67.95
          }
        }
      ]
    }
  }
}
```

7.2.4 Range Aggregation

Range Aggregation 是范围聚合，用于反映数据的分布情况。比如，对 books 索引中的图书按照价格区间在 0~50、50~80、80 以上进行范围聚合，查询语句如下：

```
POST books/_search?size=0
{
  "aggs": {
    "price_ranges": {
      "range": {
        "field": "price",
        "ranges": [
          {"to": 50},
          {"from": 50,"to": 80},
```

```
            {"from": 80}
          ]
        }
      }
    }
}
```

注意三个区间的临界值，第一段是统计价格小于 50，第二段是统计价格大于等于 50 且小于 80，第三段是价格大于等于 80。

聚合结果如下：

```
{
  "aggregations": {
    "price_ranges": {
      "buckets": [
        {
          "key": "*-50.0",
          "to": 50,
          "doc_count": 1
        },
        {
          "key": "50.0-80.0",
          "from": 50,
          "to": 80,
          "doc_count": 3
        },
        {
          "key": "80.0-*",
          "from": 80,
          "doc_count": 1
        }
      ]
    }
  }
}
```

Range Aggregation 不仅可以对数值型字段进行范围统计，也可以作用在日期类型上。例如，按图书的出版日期进行范围聚合，分为 2013 年 9 月 1 日之前、2013 年 9 月 1 日至 2014 年 9 月 1 日、2014 年 9 月 1 日之后三个区间，查询语句如下：

```
POST books/_search?size=0
{
  "aggs": {
    "range": {
      "date_range": {
        "field": "publish_time",
```

```
          "format": "yyyy-MM-dd",
          "ranges": [
            {
              "to": "2013-09-01"
            },
            {
              "from": "2013-09-01",
              "to": "2014-09-01"
            },
            {
              "from": "2014-09-01"
            }
          ]
        }
      }
    }
  }
```

聚合结果如下：

```
  "aggregations": {
    "range": {
      "buckets": [
        {
          "key": "*-2013-09-01",
          "to": 1377993600000,
          "to_as_string": "2013-09-01",
          "doc_count": 3
        },
        {
          "key": "2013-09-01-2014-09-01",
          "from": 1377993600000,
          "from_as_string": "2013-09-01",
          "to": 1409529600000,
          "to_as_string": "2014-09-01",
          "doc_count": 1
        },
        {
          "key": "2014-09-01-*",
          "from": 1409529600000,
          "from_as_string": "2014-09-01",
          "doc_count": 1
        }
      ]
    }
  }
```

7.2.5 Date Range Aggregation

Date Range Aggregation 专门用于日期类型的范围聚合，和 Range Aggregation 的区别在于日期的起止值可以使用数学表达式。例如，对 books 索引中文档的出版日期进行日期范围聚合，第一个范围在两年前，第二个范围是从两年前到现在，查询语句如下：

```
POST books/_search?size=0
{
    "aggs": {
        "range": {
            "date_range": {
                "field": "publish_time",
                "format": "yyyy-MM-dd",
                "ranges": [
                    { "to": "now-24M/M" },
                    { "from": "now-24M/M" }
                ]
            }
        }
    }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "range": {
      "buckets": [
        {
          "key": "*-2015-09-01",
          "to": 1441065600000,
          "to_as_string": "2015-09-01",
          "doc_count": 4
        },
        {
          "key": "2015-09-01-*",
          "from": 1441065600000,
          "from_as_string": "2015-09-01",
          "doc_count": 1
        }
      ]
    }
  }
}
```

7.2.6　Date Histogram Aggregation

Date Histogram Aggregation 是时间直方图聚合，常用于按照日期对文档进行统计并绘制条形图。例如，对 books 索引中的图书和出版日期按月做时间直方图聚合，聚合命令如下：

```
POST books/_search?size=0
{
    "aggs" : {
        "books_over_time" : {
            "date_histogram" : {
                "field" : "publish_time",
                "interval" : "month"
            }
        }
    }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "books_over_time": {
      "buckets": [
        {
          "key_as_string": "2007-10-01T00:00:00.000Z",
          "key": 1191196800000,
          "doc_count": 1
        }
        ......
        {
          "key_as_string": "2016-05-01T00:00:00.000Z",
          "key": 1462060800000,
          "doc_count": 1
        }
      ]
    }
  }
}
```

由于测试文档数量较少，数据分布比较稀疏，因此统计效果不明显。图 7-1 是日志处理项目中按天统计日志数量并绘制直方图的效果。

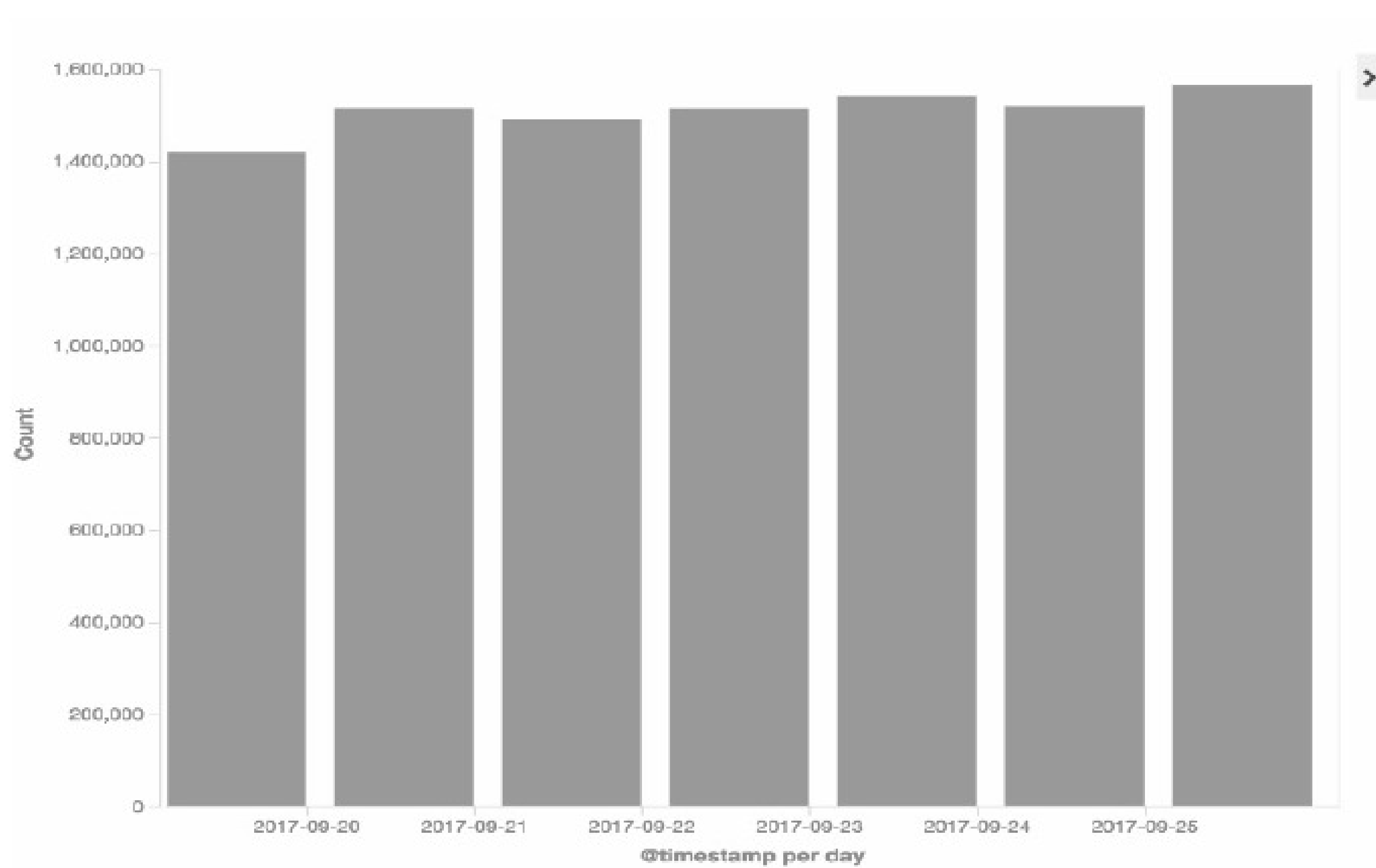


图 7-1 时间直方图聚合效果图

7.2.7 Missing Aggregation

Missing Aggregation 是空值聚合，可以把文档集中所有缺失字段的文档分到一个桶中。例如，对 books 索引中缺失 price 字段（包含取值为 null）的文档进行聚合，查询语句如下：

```
POST books/_search?size=0
{
    "aggs" : {
        "books_without_a_price" : {
            "missing" : { "field" : "price" }
        }
    }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "books_without_a_price": {
      "doc_count": 0
    }
  }
}
```

7.2.8 Children Aggregation

Children Aggregation 是一种特殊的单桶聚合，可以根据父子文档关系进行分桶。在第 6 章学习父子文档查询时，在 company 索引中指定了 employee 类型的父文档为 branch，映射如下：

```
PUT /company
{
  "mappings": {
```

```
    "branch": {},
    "employee": {"_parent": { "type": "branch"}}
  }
}
```

利用 Children Aggregation 统计子类型为 employee 的文档数量，查询语句如下：

```
POST company/_search?size=0
{
  "aggs": {
    "to-answers": {
      "children": {
        "type": "employee"
      }
    }
  }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "to-answers": {
      "doc_count": 4
    }
  }
}
```

7.2.9　Geo Distance Aggregation

Geo Distance Aggregation 用于对地理点(geo_point)做范围统计。例如，以西安为中心，分别统计距离范围在 500km 以内、500km 和 1000km 之间、1000km 以外的城市，查询语句如下：

```
POST geo/_search?size=0
{
  "aggs": {
    "city_from_xi'an": {
      "geo_distance": {
        "field": "location",
        "origin": "34.3412700000,108.9398400000",
        "unit": "km",
        "ranges": [
          {
            "to": 500
          },
          {
```

```
          "from": 500,
          "to": 1000
        },
        {
          "from": 1000
        }
      ]
    }
  }
 }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "city_from_xi'an": {
      "buckets": [
        {
          "key": "*-500.0",
          "from": 0,
          "to": 500,
          "doc_count": 2
        },
        {
          "key": "500.0-1000.0",
          "from": 500,
          "to": 1000,
          "doc_count": 2
        },
        {
          "key": "1000.0-*",
          "from": 1000,
          "doc_count": 2
        }
      ]
    }
  }
}
```

7.2.10 IP Range Aggregation

IP Range Aggregation 用于对 IP 类型数据范围聚合。例子如下：

```
POST ip_test/_search?size=0
{
```

```
    "aggs" : {
        "ip_ranges" : {
            "ip_range" : {
                "field" : "ip",
                "ranges" : [
                    { "to" : "10.0.0.5" },
                    { "from" : "10.0.0.5" }
                ]
            }
        }
    }
}
```

聚合结果如下：

```
{
  "aggregations": {
    "ip_ranges": {
      "buckets": [
        {
          "to": "10.0.0.5",
          "doc_count": 2
        },
        {
          "from": "10.0.0.5",
          "doc_count": 1
        }
      ]
    }
  }
}
```

7.3 本章小结

本章介绍了 Elasticsearch 的聚合分析功能，通过实例展示了指标聚合和桶聚合的具体用法。在实时大数据分析方面，越来越多的分布式系统使用了 Elasticsearch。

第8章

Elasticsearch Java API

本章学习要点：

- ✱ Java API 的功能
- ✱ 如何创建 Maven 工程
- ✱ 如何连接到集群
- ✱ 如何进行文档的 CRUD
- ✱ 各种搜索 API
- ✱ 使用 Java API 进行索引管理和集群管理

8.1 Java API 简介

我们在前面提到过 Elasticsearch 底层依赖于 Lucene 库，而 Lucene 库完全是 Java 编写的，RESTful API 发送的请求最后都是通过 Java 执行的。就可行性来讲，Java API 比 RESTful API 功能更强大。不论是文档的 CRUD、查询、批量操作、统计操作，还是获取集群信息、索引和集群管理，RESTful API 能做的 Java API 都能做。

所有的 Elasticsearch 操作都是通过一个客户端对象来执行的。无论是接收一个侦听器，还是返回一个结果，所有的操作都是完全异步的。除此之外，客户端上的操作可以累加，通过 Bulk 端点批量执行。我们发送的客户端请求在 Elasticsearch 内部最终都是通过 Java API 执行的。

Elasticsearch Java API 极其广泛，把所有的方法都一一介绍出来并不太现实，因此我们挑选出最常用也是最重要的 API 进行介绍，并通过数据集演示如何使用这些 API。掌握了基础之后，遇到复杂需求就能够做到举一反三、触类旁通。

8.2　Maven 依赖

Elasticsearch 依赖的 jar 包托管在 Maven 的中央仓库，在 Maven 工程的 pom.xml 文件中添加 Elasticsearch 5.4 版本的 Maven 坐标：

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>transport</artifactId>
    <version>5.4.0</version>
</dependency>
```

下面在 Eclipse 中演示如何创建 Elasticsearch 的 Maven 工程，开始前确保计算机已经正确安装 Maven。

步骤 01　在 Eclipse 中配置 Maven 路径。

如图 8-1 所示，在 Eclipse 中依次选择 Preference →Maven →Installations，Eclipse 默认有一个嵌入式的 Maven，也可以单击 Add 按钮添加本地 Maven。

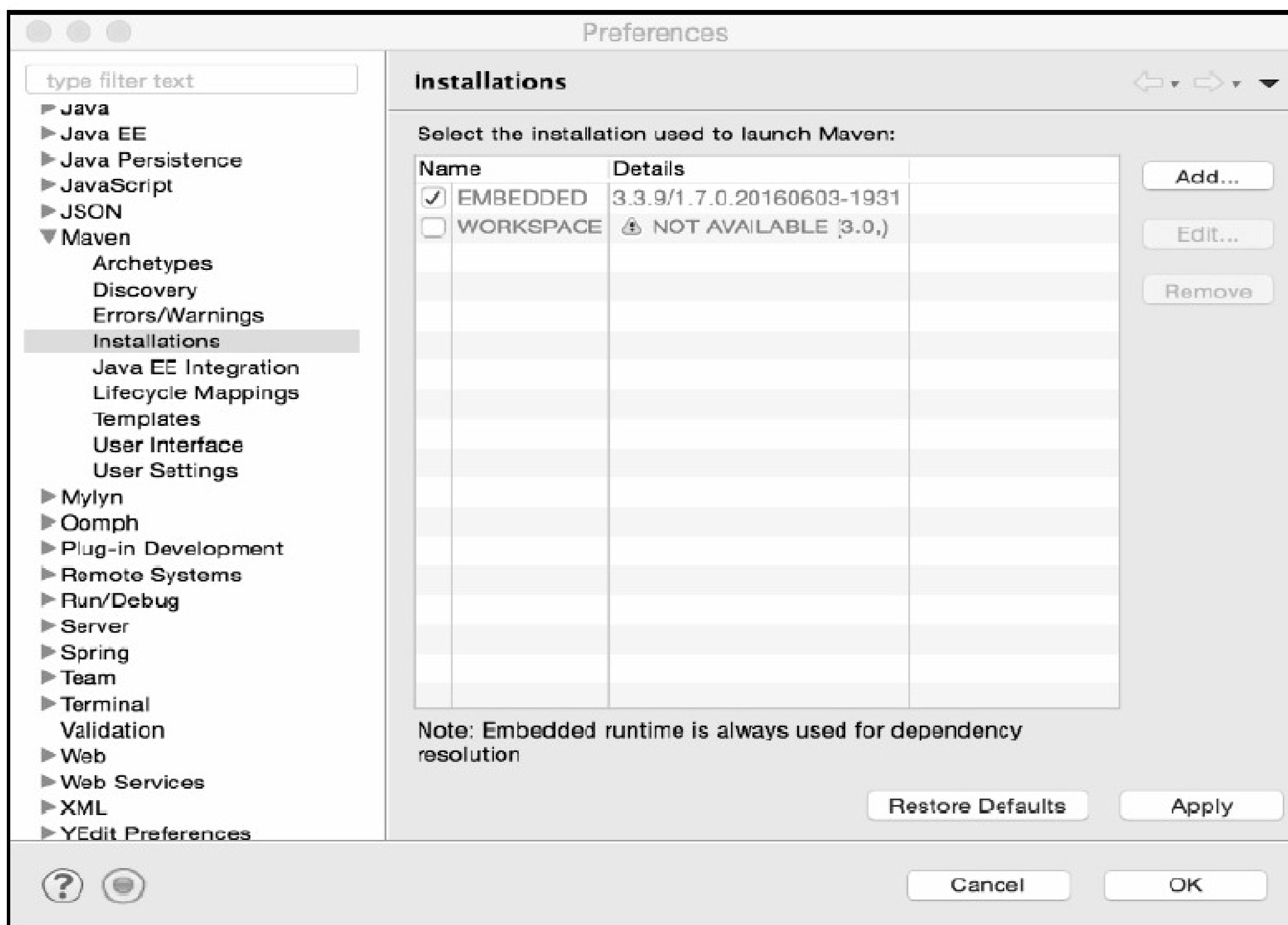


图 8-1 Eclipse 中设置 Maven 安装路径

步骤 02　新建 Maven 工程。

如图 8-2 所示，在 Eclipse 中依次选择 File →New→Other →Maven→Maven Project，然后单击 Next 按钮。

如图 8-3 所示，在新建 Maven 工程界面勾选 Create a simple project（skip archetype selection）复选框，workspace location 是新建的 Maven 项目的存储路径，可以使用默认的，也可以单击 Browse 按钮自定义。然后单击 Next 按钮进入下一步。

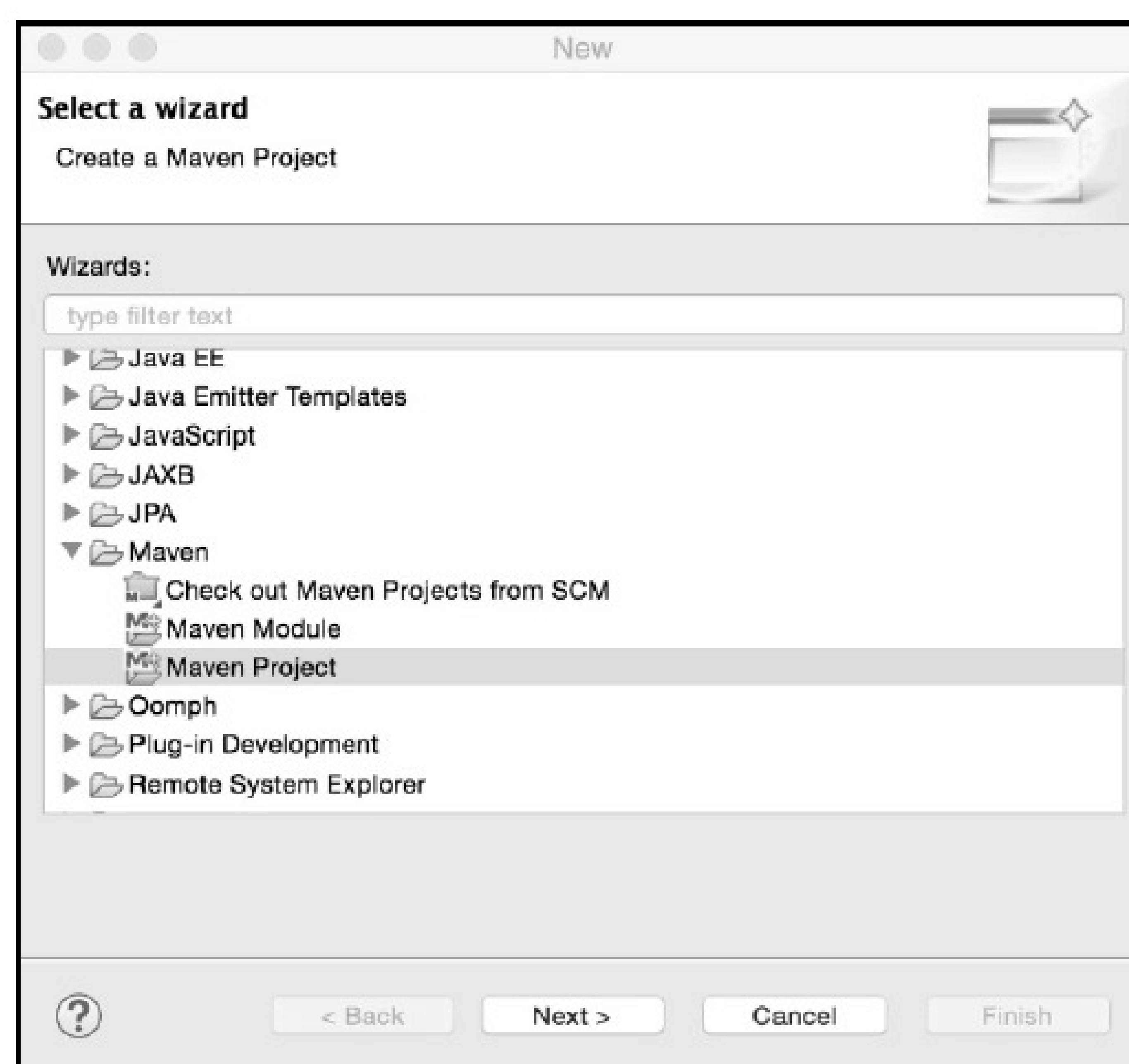


图 8-2　在 Eclipse 中创建 Maven 工程

如图 8-4 所示，这一步是配置工程的属性，Group Id 是工程的包名，Artifact Id 是工程的项目名，版本选择默认的，Packaging 选择 jar 类型，最后单击 Finish 按钮。

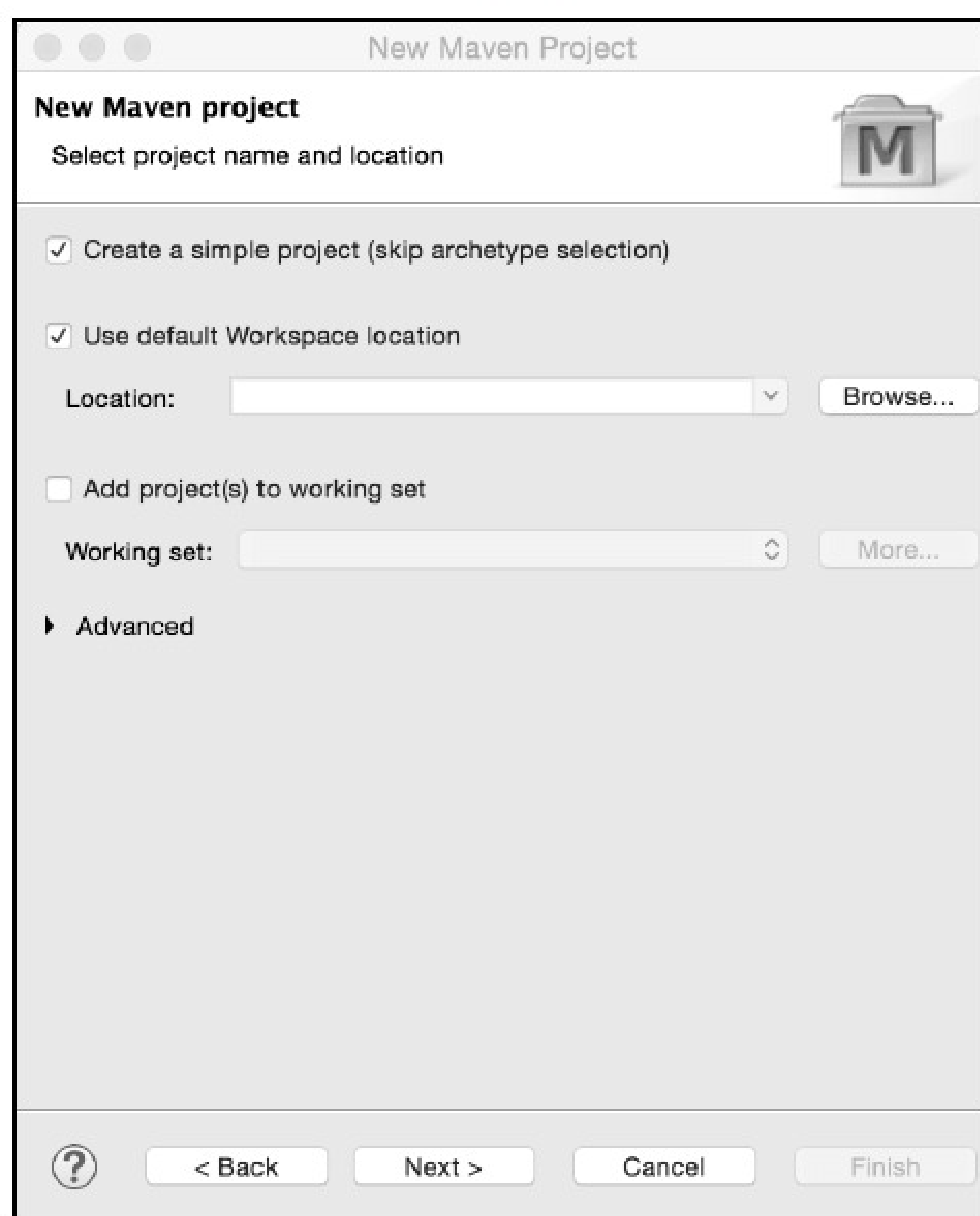


图 8-3　在 Eclipse 中创建 Maven 工程



图 8-4　在 Eclipse 中创建 Maven 工程

步骤 03　在 pom.xml 文件中添加 Elasticsearch 的外部依赖。

保存 pom.xml 文件以后 Eclipse 会自动下载 jar 包。如果 Eclipse 没有自动下载，可以单击工程名，然后右击，选择 Run As→Maven Install。jar 包下载完成以后工程路径中会多出来一个 Maven Dependencies 路径，打开以后可以看到加载的 jar 包列表。

步骤 04　日志配置。

在 pom.xml 中加入 Log4j 的 Maven 坐标：

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.8.2</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.8.2</version>
</dependency>
```

在 src/main/resources 目录下新建输出日志的配置文件 log4j2.properties，添加以下内容：

```
appender.console.type = Console
appender.console.name = console
appender.console.layout.type = PatternLayout
rootLogger.level = info
rootLogger.appenderRef.console.ref = console
```

8.3　依赖冲突

如果在已有的项目中使用 Elasticsearch，有可能会遇到第三方依赖库版本（比如 Guava、Joda）冲突的问题。比如，你的应用程序中使用的 Joda 版本是 2.1，但是 Elasticsearch 使用的是 Joda 2.8。解决包冲突问题有以下 2 个选择：

- 最简单的解决方案是升级版本。较新版本的模块更有可能修复旧版的 bug。现在使用的版本和新版本差得越远，以后升级起来会更麻烦。当然，你使用的第三方库依赖于一个过时版本的包阻止你升级也是有可能的。
- 第二个选择是重新安置有冲突的 jar 包，Java 和 Elasticsearch 保留一个。要么屏蔽 Java 应用，要么屏蔽 Elasticsearch 和 Elasticsearch 客户端所需要的任何插件。

8.4　连接到集群

Elasticsearch 的 Java client 对象可以执行多种操作：

- 在现有的群集上执行标准的 index、get、delete 和 search 操作。
- 在运行的群集上执行管理任务。

获得一个 Elasticsearch client 对象非常简单，最常用的方式是创建一个可以连接到 Elasticsearch 集群的传输机对象（TransportClient）。

需要注意的是，client 对象一定要和集群中的节点具有相同的版本，比如我们使用的

Elasticsearch 的版本是 5.4.0，那么 Java API 也要使用 5.4.0 版本。如果客户端和服务器版本不一致，就会导致有些功能无法使用，最理想的情况是客户端和服务器版本保持一致。

8.4.1　传输机连接

使用 TransportClient 创建的 client 对象可以通过传输模块远程与 Elasticsearch 集群建立连接。这种方式只会连接到集群而不会加入集群，client 对象知道一个或多个传输地址，通过轮询调度的方式和服务器交互。

创建 TransportClient 对象的方法：

```
TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)
    .addTransportAddress(new InetSocketTransportAddress(InetAddress
    .getByName ("host1"),9300))
    .addTransportAddress(new InetSocketTransportAddress(InetAddress.
    getByName("host2"), 9300));
```

Settings 对象中可以添加配置信息。如果在配置文件中设置的 Elasticsearch 集群名称不是默认的 elasticsearch，就需要在 Settings 对象中指定集群名称：

```
Settings settings = Settings.builder()
   .put("cluster.name", "myClusterName")
   .build();
TransportClient client = new PreBuiltTransportClient(settings);
```

Transportclient 对象自带集群探测功能，可以自动添加新的主机、自动移除旧的主机。如果想要打开集群探测功能，就需要设置 client.transport.sniff 的属性为 true:

```
Settings settings = Settings.builder()
   .put("client.transport.sniff", true).build();
TransportClient client = new PreBuiltTransportClient(settings);
```

更多 TransportClient 的配置如下。

- client.transport.ignore_cluster_name：设为 true 会忽略节点的集群名称验证。
- client.transport.ping_timeout：设置 ping 命令的响应时间，默认 5 秒。
- client.transport.nodes_sampler_interval：设置检查节点可用性的频率，默认值是 5 秒。

8.4.2　节点连接

节点连接的思路是把应用程序作为 Elasticsearch 的一个节点，我们的应用程序作为 Elasticsearch 集群的一部分，客户端作为一个新的节点和集群中的其他节点建立连接。这样可以减少客户端和服务器之间的交互次数，但是这种方法并不总是可行，比如集群不在同一个局域网中。推荐使用传输机方式创建 client 对象。

8.4.3　代码实现

在工程的 src/main/java 目录下新建包 tup.es.client，在该包下新建类 TestClient.java，在该类的 main 方法中创建 TransportClient 对象并进行测试。代码内容如代码清单 8-1 所示。

代码清单 8-1

```
import java.net.InetAddress;
import java.net.UnknownHostException;
import org.elasticsearch.action.get.GetResponse;
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.common.transport
      .InetSocketTransportAddress;
import org.elasticsearch.transport.client.PreBuiltTransportClient;
public class TestClient {
    public static String CLUSTER_NAME = "elasticsearch";   // 集群名称
    public static String HOST_IP = "172.31.44.9";          // 服务器 IP
    public static int TCP_PORT = 9300;// 端口
    public static void main(String[] args)
       throws UnknownHostException {
    Settings settings = Settings.builder()
             .put("cluster.name", CLUSTER_NAME)
             .build();
    TransportClient client = new PreBuiltTransportClient(settings)
            .addTransportAddress(new InetSocketTransportAddress(
                        InetAddress.getByName(HOST_IP), TCP_PORT));
    GetResponse getResponse = client
            .prepareGet("books", "IT", "1").get();
   System.out.println(getResponse.getSourceAsString());
  }
}
```

main 方法中一共有 4 行代码，第一行创建了一个 Settings 对象，在 Settings 对象指定集群的名称，第二行创建了一个 TransportClient 对象，第三行通过 TransportClient 对象的 prepareGet 方法读取 Elasticsearch 中的一个文档，返回结果存储在 GetResponse 对象中，最后一行打印了文档内容。运行结果如图 8-5 所示。

Elasticsearch Java API 的相关操作都是通过 TransportClient 对象与 Elasticsearch 集群进行交互的。为了避免每次请求都创建一个新的 TransportClient 对象，可以封装一个双重加锁单例模式返回 TransportClient 对象的方法，代码如下：

```
private volatile static TransportClient client;
public static TransportClient getSingleClient()
 throws UnknownHostException {
    if(client==null) {
        synchronized(TransportClient.class) {
            client = new PreBuiltTransportClient(settings)
                .addTransportAddress(new InetSocketTransportAddress(
```

```
                    InetAddress.getByName(HOST_IP), TCP_PORT));
            }
        }
        return client;
    }
```

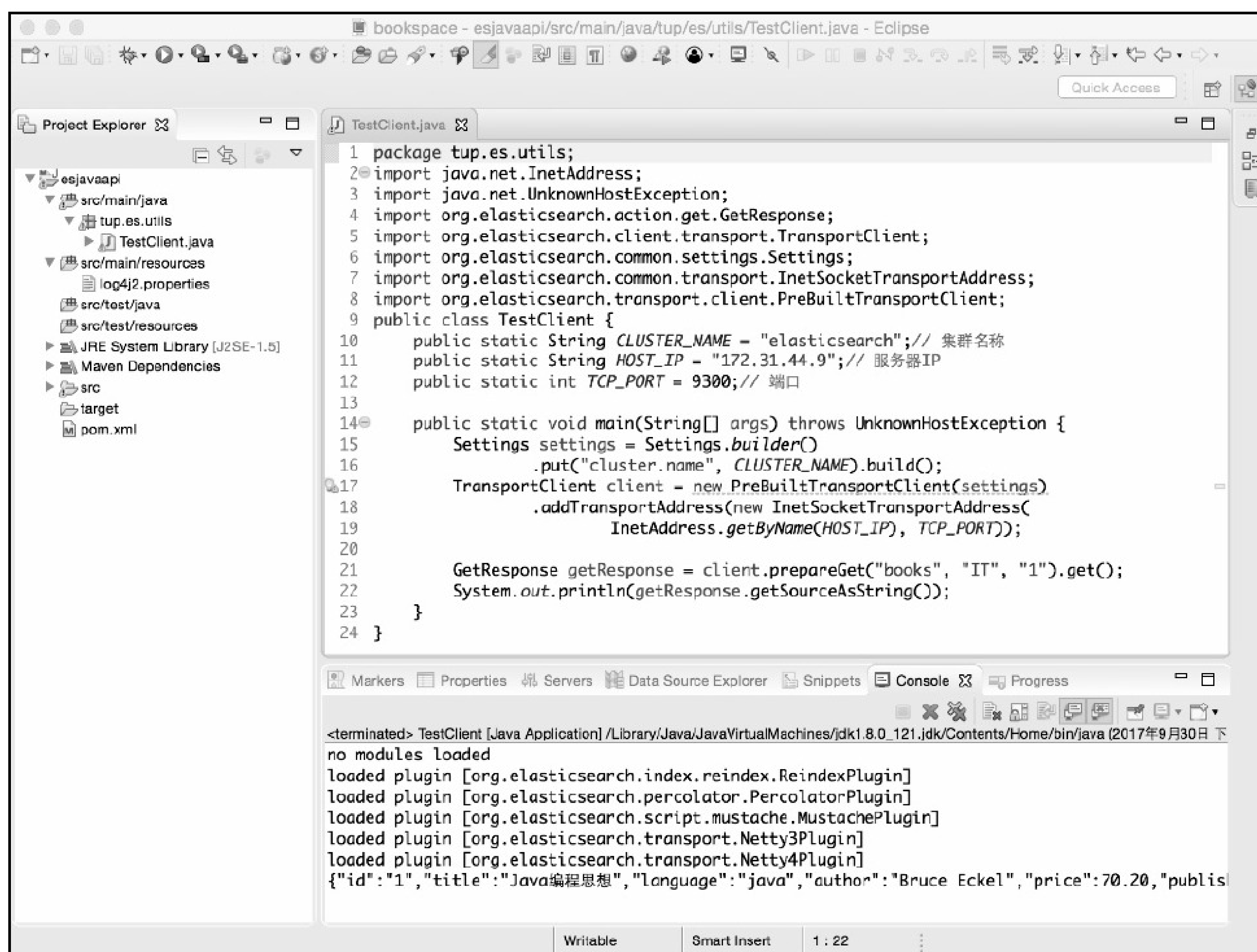


图 8-5　在 Eclipse 中测试 TransportClient 对象

8.5　索引管理

这一小节介绍如何通过 Elasticsearch Java API 进行索引的创建、删除、刷新、设置别名、设置 mapping 等索引管理操作。索引管理是通过一个 IndicesAdminClient 对象发送各种操作请求，获取 IndicesAdminClient 对象的方式如下：

```
IndicesAdminClient indicesAdminClient = client.admin().indices();
```

通过 IndicesAdminClient 对象可以执行索引管理的相关操作，简介如下。

- 判断索引是否存在

```
IndicesExistsResponse exResponse = indicesAdminClient
    .prepareExists("indexName").get();
System.out.println(exResponse.isExists());
```

● 判断 type 是否存在

```
TypesExistsResponse existsResponse=indicesAdminClient
    .prepareTypesExists("indexName")
    .setTypes("type1","type2")
    .get();
System.out.println(existsResponse.isExists());
```

● 创建一个索引

```
CreateIndexResponse cResponse = indicesAdminClient
    .prepareCreate("indexName").get();
System.out.println(cResponse.isAcknowledged());
```

注意，索引名必须小写，如果索引名不符合规范，就会出现 InvalidIndexNameException 异常。最好在创建索引之前通过 String 类的 toLowerCase()方法进行小写转换。

● 创建索引并设置 Settings

```
CreateIndexResponse cResponse = indicesAdminClient
    .prepareCreate("twitter")
    .setSettings(Settings.builder()
        .put("index.number_of_shards", 3)
        .put("index.number_of_replicas", 2)
    )
    .get();
```

● 更新副本

```
UpdateSettingsResponse upResponse = indicesAdminClient
    .prepareUpdateSettings("twitter")
    .setSettings(Settings.builder()
    .put("index.number_of_replicas", 0))
    .get();
```

● 获取 Settings

```
GetSettingsResponse response = indicesAdminClient
     .prepareGetSettings("twitter ", " tweet ").get();
for (ObjectObjectCursor<String, Settings> cursor :
    response.getIndexToSettings()) {
    String index = cursor.key;
    Settings settings = cursor.value;
    Integer shards = settings.getAsInt("index.number_of_shards",
        null);
    Integer replicas = settings.getAsInt("index.number_of_replicas",
        null);
}
```

● 设置 mapping

假设索引 twitter/tweet 的 mapping 如下：

```
{
   "properties": {
      "name": { "type": "keyword" }
   }
}
```

使用 Java API 设置 mapping 的核心代码如下：

```
client.admin().indices().preparePutMapping("twitter")
        .setType("tweet")
        .setSource("{\n" +
                "  \"properties\": {\n" +
                "    \"name\": {\n" +
                "      \"type\": \" keyword \"\n" +
                "    }\n" +
                "  }\n" +
                "}")
        . get();
```

也可以使用 XContentFactory 构造，代码如下：

```
CreateIndexResponse cResponse = indicesAdminClient
    .prepareCreate("twitter ")
    .addMapping("tweet ",XContentFactory.jsonBuilder()
    .startObject()
    .startObject("properties")
    .startObject("name")
    .field("type", " keyword ")
    .endObject().endObject()
    .endObject())
    .get();
```

● 获取 mapping

```
GetMappingsResponse mResponse = indicesAdminClient
                .prepareGetMappings("indexname").get();
ImmutableOpenMap<String, MappingMetaData> mapings = mResponse
                .getMappings()
                .get("indexname");
MappingMetaData metatda = mapings.get("typename");
System.out.println(metatda.getSourceAsMap());
```

● 删除索引

```
DeleteIndexResponse dResponse = indicesAdminClient
   .prepareDelete("indexname").get();
System.out.println(dResponse.isAcknowledged());
```

- 刷新

```
indicesAdminClient.prepareRefresh().get();
indicesAdminClient.prepareRefresh("indexname").get();
indicesAdminClient.prepareRefresh("indexname", "typename").get();
```

- 关闭索引

```
CloseIndexResponse clResponse = indicesAdminClient
    .prepareClose("indexname").get();
```

- 打开索引

```
OpenIndexResponse opResponse = indicesAdminClient
   .prepareOpen("indexname").get();
```

- 设置别名

```
IndicesAliasesResponse aResponse = indicesAdminClient
    .prepareAliases().addAlias("indexName", "aliasesName").get();
```

- 获取别名

```
GetAliasesResponse gResponse = indicesAdminClient
    .prepareGetAliases("aliasesname").get();
```

8.6　文档管理

这一节介绍文档管理的 Java API，主要包括单文档操作 API 和多文档操作 API 两部分。单文档操作 API 主要包括索引文档、查询文档、删除文档、更新文档，多文档操作 API 主要包括批量获取操作和 BULK 操作。

8.6.1　新建文档

索引文档 API 可以把一个 JSON 格式的文档索引到特定的索引中，并使该文档是可搜索的。生成 JSON 格式文档的方法有以下几种：

- 把 JSON 格式的文档手工转换为字节数组 byte[]或 String。
- 使用 Map，Map 是一个 key/value 键值对集合，代表一个 JSON 结构。
- 使用内置帮助类 XContentFactory 的 jsonBuilder()方法。
- 使用 Jackson 等第三方库把 Java Bean 转化为 JSON 序列。

事实上在内部不论哪种类型最后都会被转换成字节数组（String 字符串也会被转化成字节数组），因此如果文档已经是字节数组格式，直接使用即可，jsonbuilder 是高度优化的 JSON 生成器。

下面通过实例介绍以上 4 种索引文档的方法，准备一个 JSON 格式的文档，内容如下：

```
{
    "user":"kimchy",
    "postDate":"2013-01-30",
    "message":"trying out Elasticsearch"
}
```

方法一：把 JSON 转为 String。

```
String doc1 = "{" +
              "\"user\":\"kimchy\"," +
              "\"postDate\":\"2013-01-30\"," +
              "\"message\":\"trying out Elasticsearch\"" +
              "}";
IndexResponse response = client
            .prepareIndex("twitter", "tweet","1")
            .setSource(doc1)
            .get();
System.out.println(response.status());
```

方法二：使用 Map。

```
Map<String, Object> doc2 = new HashMap<String, Object>();
doc2.put("user","kimchy");
doc2.put("postDate","2013-01-30");
doc2.put("message","trying out Elasticsearch");
IndexResponse response = client
    .prepareIndex("twitter", "tweet","2")
    .setSource(doc2)
    .get();
System.out.println(response.status());
```

方法三：使用 Elasticsearch 帮助类。

使用内置帮助类 XContentFactory 的 jsonBuilder()方法可以构造 XContentBuilder 对象，XContentBuilder 对象可以直接写入 Elasticsearch 中。如果需要查看生成的 JSON 内容，可以调用 string()方法。

```
XContentBuilder doc3 = jsonBuilder().startObject()
    .field("user", "kimchy")
    .field("postDate", "2013-01-30")
    .field("message", "trying out Elasticsearch")
    .endObject();
System.out.println(doc3.string());
IndexResponse response = client
    .prepareIndex("twitter", "tweet", "3")
```

```
    .setSource(doc3)
    .get();
System.out.println(response.status());
```

jsonBuilder()方法也可以使用 startArray()方法添加数组，field()方法可以接收多种类型的参数，可以直接传入数字、日期甚至是其他类型的 XContentBuilder 对象。下面我们再给出一个例子，一个含有数组和嵌套对象的 JSON 文档，内容如下：

```
{
    "name":"Tom",
    "age":"12",
    "scores":[{"Math":"80"},"English":"85"],
    "address":{
       "country":"China",
      "city":"Beijing"
    }
}
```

使用 jsonBuilder()方法构造结果：

```
builder=jsonBuilder().startObject().field("name","Tom")
    .field("age","12")
    .startArray("scores")
   .startObject().field("Math","80").endObject()
    .startObject().field("English","85")
    .endObject().endArray()
    .field("address")
    .startObject().field("country","China")
    .field("city","Beijing")
    .endObject()
    .endObject();
System.out.println(builder.string());
```

方法四：使用 Jackson 序列化 Java Bean。

步骤 01 使用 Jackson 首先要加入相关依赖，把下面的配置加入 pom.xml 中：

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.6.6</version>
</dependency>
```

步骤 02 创建 User 对象，代码如下，这里不再贴出无参构造方法、有参构造方法、setter 和 getter 以及 toString 方法的代码：

```
import java.util.Date;
public class User {
    private String user;
    private Date postDate;
    private String message;
    //无参构造方法
   //有参构造方法
   // setter 和 getter
   // toString()
}
```

步骤 03 通过 ObjectMapper 序列化 Java Bean，代码如下：

```
User user = new User("Zhang San", new Date(2013 - 1900, 1 - 1, 30),
    "trying out Elasticsearch");
ObjectMapper mapper = new ObjectMapper();
SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
    mapper.setDateFormat(format);
byte[] doc4 = mapper.writeValueAsBytes(user);
IndexResponse response = client
    .prepareIndex("twitter", "tweet", "5")
    .setSource(doc4).execute().actionGet();
System.out.println(response.status());
```

最后，可以通过 IndexResponse 对象获取反馈信息，常用方法简介如下。

- 获取请求的索引名称：String _index = response.getIndex()。
- 获取请求的文档类型：String _type = response.getType()。
- 获取请求的文档 ID：String _id = response.getId()。
- 获取文档版本：long _version = response.getVersion()。
- 返回文档是否创建成功：boolean created = response.status()，如果文档是新创建的，就返回 CREATED；如果文档不是首次创建而是被更新过的就返回 OK。

8.6.2　获取文档

Get API 可以实现通过文档 id 读取一个 JSON 格式的文档。下面的例子是读取索引名为 twitter、类型名为 tweet、id 为 1 的文档：

```
GetResponse response =client.prepareGet("twitter","tweet","1")
    .get();
String content= response.getSourceAsString();
```

GetResponse 对象提供的常用方法如下。

- isExists()：如果要读取的文档存在，就返回 true，否则返回 false。
- getIndex()：返回请求文档的索引名。

- getType()：返回请求文档的类型名。
- getId()：返回请求文档的 ID。
- getVersion()：返回文档版本信息。
- getSourceAsBytes()：以二进制数组方式读取文档内容。
- getSourceAsMap()：以 map 形式读取文档内容。
- getSourceAsString()：以文本方式读取文档内容。
- isSourceEmpty()：判断文档内容是否为空。

8.6.3　删除文档

和读取文档的 API 类似，Delete API 可以实现通过文档 id 删除 Elasticsearch 中的文档，以删除索引名为 twitter、类型名为 tweet、id 为 1 的文档为例，代码如下：

```
DeleteResponse response = client.prepareDelete("twitter", "tweet", "1")
    .get();
```

DeleteResponse 对象提供的常用方法如下。

- status()：删除成功，返回 OK；删除失败，返回 NOT_FOUND。
- getType()：返回删除请求文档的类型。
- getId()：返回删除请求文档的 ID。
- getVersion()：返回删除请求文档的版本信息。

8.6.4　更新文档

Elasticsearch 提供了多种更新文档的 API，主要有使用 UpdateRequest 对象、使用内嵌脚本、使用 prepareUpdate()方法这 3 种。

给索引名为 twitter、类型名为 tweet、id 为 1 的文档新增一个 gender 字段，首先创建一个 UpdateRequest 对象，之后通过 TransportClient 对象的 Update()方法执行更新。核心代码如下：

```
UpdateRequest updateRequest = new UpdateRequest();
updateRequest.index("twitter");
updateRequest.type("tweet");
updateRequest.id("1");
updateRequest.doc(jsonBuilder()
    .startObject()
    .field("gender", "male")
    .endObject());
client.update(updateRequest).get();
```

Elasticsearch 也支持脚本操作，使用脚本给文档新增一个 gender 字段，核心代码如下：

```
UpdateRequest updateRequest = new UpdateRequest("twitter", "tweet",
    "1").script(new Script("ctx._source.gender = \"male\""));
client.update(updateRequest).get();
```

使用 prepareUpdate()方法同样可以实现更新操作，既可以通过设置文档的方式（doc 模式）

更新文档，又可以通过执行脚本的方式（script 模式）更新文档，但是这两种方式不要混用。代码示例如下：

```
client.prepareUpdate("ttl", "doc", "1")
    .setScript(new Script("ctx._source.gender = \"male\"" ,
     ScriptService.ScriptType.INLINE, null, null))
     .get();
client.prepareUpdate("ttl", "doc", "1")
    .setDoc(jsonBuilder()
    .startObject()
    .field("gender", "male")
    .endObject())
    .get();
```

Elasticsearch 还支持 upsert 操作，如果文档存在，就执行修改操作；如果文档不存在，就再创建一个新的文档。下面来看如何实现 upsert 操作：

```
IndexRequest indexRequest = new
    IndexRequest("twitter", "tweet", "1")
    .source(jsonBuilder() .startObject()
    .field("name", "Joe Smith")
    .field("gender", "male")
    .endObject());
UpdateRequest updateRequest = new
    UpdateRequest("twitter", "tweet", "1")
    .doc(jsonBuilder().startObject()
    .field("gender", "male")
    .endObject()).upsert(indexRequest);
client.update(updateRequest).actionGet();
```

如果文档 twitter/tweet/1 存在，就执行 updateRequest 操作，把 gender 修改为 male；如果文档 twitter/tweet/1 不存在，就执行 indexRequest 操作，新建一个文档。假设 Elasticsearch 存在一个文档 twitter/tweet/1：

```
{
   "name" : " Joe Dalton",
   "gender": "female"
}
```

执行上述操作以后，twitter/tweet/1 中的内容更新为：

```
{
   "name" : " Joe Dalton",
   "gender": "male"
}
```

如果 twitter/tweet/1 不存在，执行上述操作以后，我们会得到一个新的文档：

```
{
    "name" : "Joe Smith",
    "gender": "male"
}
```

8.6.5 查询删除

Delete By Query API 可以实现根据查询条件删除文档，删除 books 索引中 title 字段包含关键词 java 的文档，代码如下：

```
BulkByScrollResponse response =DeleteByQueryAction.INSTANCE
    .newRequestBuilder(client)                              //注释 1
    .filter(QueryBuilders.matchQuery("title", "java"))      //注释 2
    .source("books")                                        //注释 3
    .get();
long deleted = response.getDeleted();                       //注释 4
```

- 注释 1：传入 TransportClient 对象。
- 注释 2：传入删除的 Query。
- 注释 3：设置索引名称。
- 注释 4：被删除文档的数目。

8.6.6 批量获取

使用 multiGet API 可以通过索引名、类型名、文档 id 一次获取一个文档集合，文档可以来自同一个索引库，也可以来自不同索引库。核心代码如下：

```
MultiGetResponse multiGetItemResponses = client.prepareMultiGet()
    .add("twitter", "tweet", "1")                           //注释 1
    .add("twitter", "tweet", "2", "3", "4")                 //注释 2
    .add("another", "type", "foo")                          //注释 3
    .get();
for (MultiGetItemResponse itemResponse : multiGetItemResponses){   //注释 4
    GetResponse response = itemResponse.getResponse();
    if (response !=null&&response.isExists()) {             //注释 5
        String json = response.getSourceAsString();         //注释 6
        System.out.println(json);
    }
}
```

- 注释 1：通过单一的 id 获取一个文档。
- 注释 2：传入多个 id，从相同的索引名/类型名中获取多个文档。
- 注释 3：可以同时获取不同索引中的文档。

- 注释 4：遍历结果集。
- 注释 5：检验文档是否存在。
- 注释 6：获取源文档。

8.6.7 批量操作

使用 multiGet API 可以进行批量读取操作，同样，使用 Bulk API 可以通过一次请求完成批量索引文档、批量删除文档和批量更新文档。下面的代码先创建了一个 BulkRequestBuilder 对象用于执行批量操作，使用 IndexRequestBuilder 创建了一个索引文档请求对象，使用 DeleteRequestBuilder 创建了一个删除文档请求对象，使用 UpdateRequestBuilder 创建了一个索引文档请求对象，最后通过调用 add()方法把 3 个请求加到 BulkRequestBuilder 对象中并批量执行。

```
BulkRequestBuilder bulkRequest = client.prepareBulk();
IndexRequestBuilder indexRequest = client
   .prepareIndex("twitter", "tweet", "5")
    .setSource(jsonBuilder().startObject().field("user", "kimchy")
    .field("postDate", new Date())
    .field("message", "another post").endObject());
DeleteRequestBuilder deleteRequest = client
    .prepareDelete("twitter","tweet", "2");
UpdateRequestBuilder updateRequest=client
    .prepareUpdate("twitter", "tweet", "5")
    .setDoc(jsonBuilder().startObject()
    .field("message", "update request")
    .endObject());
bulkRequest.add(indexRequest).add(deleteRequest).add(updateRequest)
    .execute().actionGet();
```

Elasticsearch 的 Bulk Processor API 可以在批量操作完成之前和之后进行相应的操作，示例代码与注释如下：

```
Listener listener = new BulkProcessor.Listener() {
    @Override
    public void beforeBulk(long arg0, BulkRequest arg1)
    {                                       //注释 1
    }
    @Override
    public void afterBulk(long arg0, BulkRequest arg1,
     Throwable arg2) {                      //注释 2
    }
    @Override
    public void afterBulk(long arg0, BulkRequest arg1,
    BulkResponse arg2)                      //注释 3
    {
```

```
    }
};
BulkProcessor bulkProcessor = BulkProcessor
        .builder(client,listener)  .setBulkActions(10000)          //注释 4
        .setBulkSize(new ByteSizeValue(20,ByteSizeUnit.MB))        //注释 5
        .setFlushInterval(TimeValue.timeValueSeconds(5))           //注释 6
        .setConcurrentRequests(5)                                  //注释 7
        .setBackoffPolicy(BackoffPolicy.exponentialBackoff(TimeValue
        .timeValueMillis(100), 3)).build();                        //注释 8
```

- 注释 1：beforeBulk()会在批量提交之前执行，通过 BulkRequest 对象的 requests()方法获取请求信息，BulkRequest 对象的 numberOfActions()方法获取请求数量。
- 注释 2：设置 bulk 批处理请求出现异常需要执行哪些操作。
- 注释 3：设置 bulk 批处理请求执行成功之后需要执行哪些操作。
- 注释 4：设置请求操作的数量超过 1 万次触发批量提交动作。
- 注释 5：设置批处理请求达到 20M 触发批量提交动作。
- 注释 6：设置刷新索引时间间隔。
- 注释 7：设置并发处理线程个数。
- 注释 8：设置回滚策略，等待时间为 100ms，retry 次数为 3 次。

8.7　搜索详解

和 REST 接口的查询 DSL 一样，Elasticsearch 也提供了 Java 接口的查询 DSL。构造查询对象的工厂类是 QueryBuilders，只要查询语句准备好了就可以使用搜索相关的 API。这一节我们首先会给出一个 match 查询的例子，通过这个例子介绍创建查询语句、获取搜索结果以及实现搜索高亮。举一反三，后面我们只给出构造其他查询的方法，把例子中的 match 查询替换掉就可以实现多种类型的搜索。基本查询和聚合分析我们使用第 6 章中的 books 索引作为测试数据集，父子文档搜索我们使用第 6 章中的 company 索引作为测试数据集。

首先构造一个 match 查询的对象：

```
QueryBuilder matchQuery = QueryBuilders.matchQuery("title",
 "Java 编程").operator(Operator.AND);
```

第一个参数是查询的字段，第二个参数是要查询的关键字，Operator.AND 表示使用 AND 的方式连接被解析后的词项。完整的代码如代码清单 8-2 所示。

代码清单 8-2

```
public class EsMatchQueryTest {
   public static void main(String[] args)
      throws UnknownHostException {
```

```
        QueryBuilder matchQuery = QueryBuilders
               .matchQuery("title", "python")
               .operator(Operator.AND);                              //注释 1
        HighlightBuilder highlighter = new HighlightBuilder()     //注释 2
                .field("title")                                      //注释 3
                .preTags("<span style=\"color:red\">")               //注释 4
                .postTags("</span>");

         SearchResponse response = EsUtils.getSingleTransportClient()
                .prepareSearch("books")                          //注释 5
                .setQuery(matchQuery)                            //注释 6
                .highlighter(highlighter)                        //注释 7
                .setSize(100)                                    //注释 8
                .get();
         SearchHits hits = response.getHits();                   //注释 9
         System.out.println("共搜索到:" + hits.getTotalHits() + "条数据");
          for (SearchHit hit : hits) {                           //注释 10
             System.out.println("Soucrce:" +hit.getSourceAsString()); //注释 11
             System.out.println("Soucrce As Map:"+ hit.getSource());  //注释 12
             System.out.println("Index:" + hit.getIndex());           //注释 13
             System.out.println("Type:" + hit.getType());             //注释 14
             System.out.println("ID:" + hit.getId());                 //注释 15
             System.out.println("Price:" + hit.getSource()
               .get("price")); //注释 16
             System.out.println("Score:" + hit.getScore());       //注释 17
         Text[] text = hit.getHighlightFields().get("title")
            .getFragments();//注释 18
         if (text != null) {
           for (Text str : text) {
                System.out.println(str.string());
           }
         }
       }
     }
 }
```

上面的代码中先创建了一个 TransportClient 对象用于和 Elasticsearch 服务器交互，之后调用 matchQuery()方法创建一个 QueryBuilder 对象，然后通过 client 对象进行查询，查询结果通过 SearchResponse 对象的 getHits()方法获取，最后遍历查询结果集输出文档内容，注释如下。

- 注释 1：构造一个 matchQuery，matchQuery 接收 2 个参数，第一个参数是要搜索的字段，第二参数用于查询字符串。Operator.AND 参数用于指定查询语句被解析后采用 AND 的方式连接。

- 注释 2：构造一个 HighlightBuilder 对象。
- 注释 3：设置要高亮的字段。
- 注释 4：自定义高亮标签。
- 注释 5：通过 TransportClient 对象调用 prepareSearch 方法，方法的参数是要检索的索引名。如果要搜索多个索引，就可以用逗号隔开，例如搜索 books1 和 books2 这两个索引，可以写成 client.prepareSearch("books1","books2")。如果不指定索引名，就会搜索集群中的所有索引。搜索结果存在于 SearchResponse 对象中。
- 注释 6：设置查询方法，参数为上面构造的 matchQuery。
- 注释 7：传入 HighlightBuilder 对象。
- 注释 8：设置一次查询返回文档的数量。
- 注释 9：通过 SearchResponse 对象的 getHits()方法返回搜索结果。
- 注释 10：遍历 SearchHits 数组。
- 注释 11：getSourceAsString()方法会返回 String 类型的文档内容。
- 注释 12：getSource()方法会返回 Map 格式的文档内容。
- 注释 13：getIndex()方法返回文档所在的索引。
- 注释 14：getType()方法返回文档所在的类型。
- 注释 15：getId()方法会返回文档的 ID。
- 注释 16：getSource()方法返回的是一个 Map，通过 get()方法获取字段的 value。
- 注释 17：getScore()方法会返回文档的评分。
- 注释 18：getHighlightFields()会返回文档中所有高亮字段的高亮内容，再通过 get()方法获取某一个字段的高亮片段，最后调用 getFragments()方法，返回结果存在于 Text 类型的数组中，遍历数组获取高亮内容。

图 8-6 是运行之后在控制台中的打印结果。

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console  Progress  JUnit
<terminated> EsMatchQueryTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java (2017年9月30日 下午11:38:14)
no modules loaded
loaded plugin [org.elasticsearch.index.reindex.ReindexPlugin]
loaded plugin [org.elasticsearch.percolator.PercolatorPlugin]
loaded plugin [org.elasticsearch.script.mustache.MustachePlugin]
loaded plugin [org.elasticsearch.transport.Netty3Plugin]
loaded plugin [org.elasticsearch.transport.Netty4Plugin]
共搜索到:2条数据
Soucrce:{"id":"4","title":"Python基础教程","language":"python","author":"Helant","price":54.50,"publish_time
Soucrce As Map:{author=Helant, price=54.5, publish_time=2014-03-01, description=经典的Python入门教程，层次鲜明,
Index:books
Type:IT
ID:4
Price:54.5
Score:1.0131714
</span>Python</em>基础教程
Soucrce:{"id":"3","title":"Python科学计算","language":"python","author":"张若愚","price":81.40,"publish_time"
Soucrce As Map:{author=张若愚, price=81.4, publish_time=2016-05-01, description=零基础学python,光盘中作者独家整
Index:books
Type:IT
ID:3
Price:81.4
Score:0.6099695
</span>Python</em>科学计算
```

图 8-6　match query 查询结果

8.7.1 全文查询

使用 Elasticsearch Java API 构造各种全文级别查询的例子如下。

- Match All Query

```
QueryBuilder matchAllQuery = QueryBuilders.matchAllQuery();
```

- match_phrase query

```
QueryBuilder matchPhraseQuery=QueryBuilders
    .matchPhraseQuery("foo","hello world");
```

- match_phrase_prefix query

```
QueryBuilder matchPhrasePrefixQuery=QueryBuilders
    .matchPhrasePrefixQuery("foo","hello w");
```

- multi_match query

```
QueryBuilder multiMatchQuery = QueryBuilders
    .multiMatchQuery("kimchy ", "user", "message" );
```

- common_terms query

```
QueryBuilder commonTermsQuery = QueryBuilders
    .commonTermsQuery("name","kimchy");
```

- query_string query

```
QueryBuilder queryStringQuery = QueryBuilders
    .queryStringQuery("+kimchy -elasticsearch");
```

- simple_query_string

```
QueryBuilder qb = QueryBuilders
    .simpleQueryStringQuery("+kimchy -elasticsearch");
```

8.7.2 词项查询

使用 Elasticsearch Java API 构造各种词项级别查询的例子如下。

- term query

```
QueryBuilder termQuery=QueryBuilders.termQuery("title","java");
```

- terms query

```
QueryBuilder termsQuery=QueryBuilders
   .termsQuery("title","java","python");
```

- range query

```
QueryBuilder rangeQuery=QueryBuilders.rangeQuery("price")
            .from(50)
```

```
             .to(70)
             .includeLower(true)
             .includeUpper(false);
```

- exists query

```
QueryBuilder existsQuery = QueryBuilders.existsQuery("language");
```

- prefix query

```
QueryBuilder prefixQuery=QueryBuilders
   .prefixQuery("description","win");
```

- wildcard query

```
QueryBuilder wildcardQuery = QueryBuilders
   .wildcardQuery("author","张若?");
```

- regexp query

```
QueryBuilder regexpQuery = QueryBuilders.regexpQuery("author", "Br.*");
```

- fuzzy query

```
QueryBuilder fuzzyQuery = QueryBuilders
   .fuzzyQuery("title","javascritp");
```

- type query

```
QueryBuilder typeQuery = QueryBuilders.typeQuery("IT");
```

- ids query

```
QueryBuilder idsQuery = QueryBuilders.idsQuery().ids("3", "5");
```

8.7.3　复合查询

使用 Elasticsearch Java API 构造各种复合查询的例子如下。

- constant_score query

```
 QueryBuilder constantScoreQuery =QueryBuilders.constantScoreQuery(
             QueryBuilders.termQuery("title","java")
      ).boost(2.0f);
```

- dis_max query

```
QueryBuilder disMaxQuery =QueryBuilders.disMaxQuery()
             .add(QueryBuilders.termQuery("title", "java"))
             .add(QueryBuilders.termQuery("title", "python"))
             .boost(1.2f)
             .tieBreaker(0.7f);
```

- bool query

使用 bool 查询查找 title 字段中包含关键词 java，并且价格不高于 70，description 字段可以包含也可以不包含虚拟机的书籍，构造 boolQuery 的代码如下：

```
QueryBuilder matchQuery1 = QueryBuilders
            .matchQuery("title", "Java");
QueryBuilder matchQuery2 = QueryBuilders
            .matchQuery("description", "虚拟机");
QueryBuilder rangeQuery= QueryBuilders.rangeQuery("price").gte(70);
QueryBuilder boolQuery = QueryBuilders.boolQuery()
            .must(matchQuery1)
            .should(matchQuery2)
            .mustNot(rangeQuery);
```

- indices query

使用 indices 查询查找索引 books 和 books2 中 title 字段包含关键词 javascript、其他索引的 message 字段包含关键词 Elasticsearch 的文档，构造 indicesQuery 的代码如下：

```
QueryBuilder matchQuery=QueryBuilders
            .matchQuery("title", "javascript");
QueryBuilder noMatchQuery=QueryBuilders
            .matchQuery("message", "Elasticsearch");
QueryBuilder indicesQuery=QueryBuilders
            .indicesQuery(matchQuery, "books","books2")
            .noMatchQuery(noMatchQuery);
```

- function_score query

```
import static org.elasticsearch.index.query.functionscore
.ScoreFunctionBuilders.*;
FilterFunctionBuilder[] functions = {
      new FunctionScoreQueryBuilder.FilterFunctionBuilder(
             matchQuery("name", "kimchy"),
             randomFunction("ABCDEF")),
      new FunctionScoreQueryBuilder.FilterFunctionBuilder(
             exponentialDecayFunction("age", 0L, 1L))
};
QueryBuilder qb = QueryBuilders.functionScoreQuery(functions);
```

- boosting query

使用 boosting 查询查找 title 字段包含关键词 python 的书籍，并对出版日期在 2015 年之前的结果降低评分，构造 boostingQuery 的代码如下：

```
QueryBuilder matchQuery=QueryBuilders.matchQuery("title", "python");
```

```
QueryBuilder rangeQuery=QueryBuilders.rangeQuery("publish_time")
    .lte("2015-01-01");
QueryBuilder boostingQuery = QueryBuilders.boostingQuery(matchQuery,
    rangeQuery).negativeBoost(0.2f);
```

8.7.4 嵌套查询

- nested query

```
QueryBuilder qb = nestedQuery(
       "obj1",
       boolQuery()
              .must(matchQuery("obj1.name", "blue"))
              .must(rangeQuery("obj1.count").gt(5)),
       ScoreMode.Avg
 );
```

- hasChildQuery 搜索含有 1980 年以后出生的员工所在的分支机构

```
QueryBuilder rangeQuery = QueryBuilders.rangeQuery("dob")
    .gte("1980-01-01");
QueryBuilder hasChildQuery = QueryBuilders.hasChildQuery("employee",
    rangeQuery);
```

- hasChildQuery 搜索最少含有 2 个 employee 的机构

```
QueryBuilder matchAllQuery = QueryBuilders.matchAllQuery();
QueryBuilder hasChildQuery = QueryBuilders
    .hasChildQuery("employee", matchAllQuery).minChildren(2);
```

- hasParentQuery 搜索哪些 employee 工作在 UK

```
QueryBuilder matchQuery2 = QueryBuilders
    .matchQuery("country", "UK");
QueryBuilder hasParentQuery = QueryBuilders
    .hasParentQuery("branch",matchQuery2);
```

8.7.5 位置查询

Java API 中执行地理位置查询，需要 spatial4j 和 jts 库的支持，在工程中添加它们的 maven 坐标如下：

```
<dependency>
   <groupId>org.locationtech.spatial4j</groupId>
   <artifactId>spatial4j</artifactId>
   <version>0.6</version>
</dependency>

<dependency>
```

```
    <groupId>com.vividsolutions</groupId>
    <artifactId>jts</artifactId>
    <version>1.13</version>
    <exclusions>
        <exclusion>
            <groupId>xerces</groupId>
            <artifactId>xercesImpl</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

- geo_shape query

```
Coordinate topLeft = new Coordinate(106.23248, 38.48644);
Coordinate bottomRight = new Coordinate(115.85794, 28.68202);
QueryBuilder geoShapeQuery = null;
geoShapeQuery = QueryBuilders.geoShapeQuery(
     "location",ShapeBuilders.newEnvelope(topLeft,bottomRight))
     .relation(ShapeRelation.WITHIN);
```

- geo_bounding_box query

```
QueryBuilder geoBoundingBoxQuery=QueryBuilders
   .geoBoundingBoxQuery("location")
   .setCorners(38.4864400000, 106.2324800000,
   28.6820200000, 115.8579400000);
```

- geo_distance query

```
QueryBuilder geoDistanceQuery =QueryBuilders
   .geoDistanceQuery("location")
   .point(39.0851000000,117.1993700000)
   .distance(200, DistanceUnit.KILOMETERS);
```

- geo_polygon query

```
List<GeoPoint> points = new ArrayList<GeoPoint>();
points.add(new GeoPoint(40.8414900000, 111.7519900000));
points.add(new GeoPoint(29.5647100000, 106.5507300000));
points.add(new GeoPoint(31.2303700000, 121.4737000000));
QueryBuilder geoPolygonQuery =QueryBuilders
     .geoPolygonQuery("location", points);
```

8.7.6　特殊查询

- more_like_this query

```
String[] fields = {"title", "description"};
String[] texts = {"python"};
MoreLikeThisQueryBuilder.Item[] items = null;
QueryBuilder moreLikeThisQuery =QueryBuilders
```

```
    .moreLikeThisQuery(fields, texts, items)
    .minTermFreq(1)
    .maxQueryTerms(12);
```

- script query

```
QueryBuilder scriptQuery =QueryBuilders.scriptQuery(
              new Script("doc['price'].value > 80")
);
```

- percolate query

设置 mapping：

```
client.admin().indices().prepareCreate("my_index")
    .addMapping("queries", "query", "type=percolator")
    .addMapping("laptop", "price", "type=long","name","type=text")
    .get();
```

注册 query：

```
QueryBuilder boolQuery = QueryBuilders.boolQuery()
              .must(QueryBuilders.rangeQuery("price").lte(10000))
              .must(QueryBuilders.matchQuery("name","macbook"));
client().prepareIndex("my_index", "queries", "1")
           .setSource(jsonBuilder()
           .startObject()
           .field("query", boolQuery)
           .endObject())
           .setRefreshPolicy(WriteRequest.RefreshPolicy.IMMEDIATE)
                 .get();
```

执行查询：

```
XContentBuilder docBuilder = XContentFactory.jsonBuilder()
                 .startObject()
                 .field("price", 9999)
                 .field("name","macbook on sale")
                 .endObject();
PercolateQueryBuilder percolateQuery = new PercolateQueryBuilder("query",
    "laptop", docBuilder.bytes());
```

8.8 聚合分析

这一节介绍聚合分析的 Java API，首先通过例子给出求 books 索引中图书价格最大值的方法，代码如代码清单 8-3 所示。

代码清单 8-3　聚合查询最大值

```
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.search.aggregations.AggregationBuilders;
import org.elasticsearch.search.aggregations.metrics.max.Max;
import org.elasticsearch.search.aggregations.metrics.max
        .MaxAggregationBuilder;

public class MaxAggregationTest {
    public static void main(String[] args) {
        MaxAggregationBuilder aggregation =
                AggregationBuilders
                        .max("agg")
                        .field("price");                       //注释 1
        SearchResponse sr = client().prepareSearch("books")    //注释 2
                .addAggregation(aggregation)                   //注释 3
                .get();
        Max agg = sr.getAggregations().get("agg");             //注释 4
        double value = agg.getValue();                         //注释 5
        System.out.println(value);
    }
}
```

- 注释 1：使用 AggregationBuilders 创建一个求最大值的聚合查询，聚合字段为 price。
- 注释 2：设置要搜索的索引名。
- 注释 3：调用 addAggregation 方法。
- 注释 4：通过 SearchResponse 对象返回聚合结果。
- 注释 5：获取最终的聚合结果。

8.8.1　指标聚合

求最小值、求和、求平均值、基本统计、高级统计、基数统计、百分位统计的核心代码如下。

- Min Aggregation

```
MinAggregationBuilder minAgg = AggregationBuilders.min("agg")
    .field("price");
SearchResponse  response =client
                .prepareSearch("books").addAggregation(minAgg)
                .execute().actionGet();
Min min = response.getAggregations().get("agg");
double minValue = min.getValue();
System.out.println(minValue);
```

- Sum Aggregation

```
SumAggregationBuilder sumAgg = AggregationBuilders.sum("agg")
    .field("price");
SearchResponse response = client().prepareSearch("books")
    .addAggregation(sumAgg).execute().actionGet();
Sum sumvalue = response.getAggregations().get("agg");
System.out.println(sumvalue.getValue());
```

- Avg Aggregation

```
AvgAggregationBuilder avgAgg = AggregationBuilders.avg("agg")
    .field("price");
SearchResponse response = client().prepareSearch("books")
    .addAggregation(avgAgg).execute().actionGet();
Avg avg = response.getAggregations().get("agg");
double avgValue = avg.getValue();
System.out.println(avgValue);
```

- Stats Aggregation

```
StatsAggregationBuilder statsAgg = AggregationBuilders.stats("agg")
    .field("price");
SearchResponse  response = client().prepareSearch("books")
    .addAggregation(statsAgg).execute().actionGet();
Stats statsValues = response.getAggregations().get("agg");
System.out.println(statsValues.getMin());
System.out.println(statsValues.getMax());
System.out.println(statsValues.getAvg());
System.out.println(statsValues.getSum());
System.out.println(statsValues.getCount());
```

- Extended Stats Aggregation

```
ExtendedStatsAggregationBuilder extendedStatsAgg =
AggregationBuilders.extendedStats("agg").field("price");
SearchResponse  response = client().prepareSearch("books")
      .addAggregation(extendedStatsAgg).execute().actionGet();
ExtendedStats extendedStatsValue = response.getAggregations()
      .get("agg");
System.out.println(extendedStatsValue.getMin());
System.out.println(extendedStatsValue.getMax());
System.out.println(extendedStatsValue.getAvg());
System.out.println(extendedStatsValue.getSum());
System.out.println(extendedStatsValue.getStdDeviation());
System.out.println(extendedStatsValue.getSumOfSquares());
System.out.println(extendedStatsValue.getVariance());
```

- Cardinality Aggregation

```
CardinalityAggregationBuilder cardAgg = AggregationBuilders
    .cardinality("agg").field("language");
SearchResponse  response = client().prepareSearch("books")
    .addAggregation(cardAgg).execute().actionGet();
Cardinality cardValue = response.getAggregations().get("agg");
System.out.println(cardValue.getValue());
```

- Percentiles Aggregation

```
PercentilesAggregationBuilder percentAgg = AggregationBuilders
    .percentiles("agg").field("price");
SearchResponse  response = client().prepareSearch("books")
    .addAggregation(percentAgg).execute().actionGet();
Percentiles percentValue = response.getAggregations().get("agg");
for (Percentile entry : percentValue) {
        double percent = entry.getPercent();
        double pValue = entry.getValue();
        System.out.printf("percent [%f], value [{%f}]", percent,
        pValue);
}
```

- Value Count Aggregation

```
ValueCountAggregationBuilder aggregation = AggregationBuilders
    .count("agg").field("author");
SearchResponse  response = client().prepareSearch("books")
    .addAggregation(aggregation).execute().actionGet();
ValueCount agg = response.getAggregations().get("agg");
long value = agg.getValue();
System.out.println(value);
```

8.8.2　桶聚合

- Terms Aggregation

```
TermsAggregationBuilder termAgg = AggregationBuilders
    .terms("per_count").field("language");
SearchResponse response = EsUtils.getSingleTransportClient()
    .prepareSearch("books").addAggregation(termAgg)
    .execute().actionGet();
Terms genders = response.getAggregations().get("per_count");
for (Terms.Bucket entry : genders.getBuckets()) {
    System.out.println(entry.getKey()+"---"+entry.getDocCount());
}
```

- Filter Aggregation

```
FilterAggregationBuilder filterAgg = AggregationBuilders
    .filter("agg", QueryBuilders.termQuery("title", "java"));
SearchResponse response =client().prepareSearch("books")
    .addAggregation(filterAgg).execute().actionGet();
Filter agg = response.getAggregations().get("agg");
System.out.println(agg.getDocCount());
```

- Filters Aggregation

```
 AggregationBuilder filtersAgg =AggregationBuilders
    .filters("agg",new FiltersAggregator.KeyedFilter("java",
    QueryBuilders.termQuery("title", "java")),new
    FiltersAggregator.KeyedFilter("python",
    QueryBuilders.termQuery("title", "python")));
 SearchResponse response = client().prepareSearch("books")
    .addAggregation(filtersAgg).execute().actionGet();
 Filters agg = response.getAggregations().get("agg");
 for (Filters.Bucket entry : agg.getBuckets()) {
      String key = entry.getKeyAsString();
      long docCount = entry.getDocCount();
      System.out.println(key + "---" + docCount);
 }
```

- Range Aggregation

```
  AggregationBuilder rangeAgg =AggregationBuilders
     .range("agg")
     .field("price")
     .addUnboundedTo(50)
     .addRange(50, 80)
     .addUnboundedFrom(80);
  SearchResponse response = client().prepareSearch("books")
     .addAggregation(rangeAgg).execute().actionGet();
  Range agg = response.getAggregations().get("agg");
  for (Range.Bucket entry : agg.getBuckets()) {
       String key = entry.getKeyAsString();
       Number from = (Number) entry.getFrom();
       Number to = (Number) entry.getTo();
       long docCount = entry.getDocCount();
       System.out.println(key+"--"+docCount);
  }
```

- Date Range Aggregation

```
AggregationBuilder dateAgg = AggregationBuilders
    .dateRange("agg")
```

```
    .field("publish_time")
    .format("yyyy-MM-dd")
    .addUnboundedTo("now-24M/M")
    .addUnboundedFrom("now+24M/M");
SearchResponse response = client().prepareSearch("books")
      .addAggregation(dateAgg).execute().actionGet();
Range agg = response.getAggregations().get("agg");
for (Range.Bucket entry : agg.getBuckets()) {
     String key = entry.getKeyAsString();
     DateTime fromAsDate = (DateTime) entry.getFrom();
     DateTime toAsDate = (DateTime) entry.getTo();
     long docCount = entry.getDocCount();
     System.out.println(key+"--"+docCount);// Doc count
}
```

- Date Histogram Aggregation

```
AggregationBuilder dateHisAgg =AggregationBuilders
   .dateHistogram("agg")
   .field("publish_time")
   .dateHistogramInterval(DateHistogramInterval.YEAR);
SearchResponse response =client().prepareSearch("books")
   .addAggregation(dateHisAgg)
   .execute().actionGet();
Histogram agg = response.getAggregations().get("agg");
for (Histogram.Bucket entry : agg.getBuckets()) {
    DateTime key = (DateTime) entry.getKey();
    String keyAsString = entry.getKeyAsString();
    long docCount = entry.getDocCount();
    System.out.println(key+"--"+docCount);
}
```

- Missing Aggregation

```
MissingAggregationBuilder missAgg = AggregationBuilders
    .missing("agg").field("price");
SearchResponse response =client().prepareSearch("books")
    .addAggregation(missAgg).execute().actionGet();
Missing agg = response.getAggregations().get("agg");
System.out.println(agg.getDocCount());
```

- Children Aggregation

```
AggregationBuilder childrenAgg =AggregationBuilders
    .children("agg", "employee");
SearchResponse response = client().prepareSearch("company")
    .addAggregation(childrenAgg).execute().actionGet();
```

```
Children agg = response.getAggregations().get("agg");
System.out.println(agg.getDocCount());
```

- Geo Distance Aggregation

```
AggregationBuilder geoDistanceAgg =AggregationBuilders
    .geoDistance("agg", new GeoPoint(34.3412700000, 108.9398400000))
    .field("location")
    .unit(DistanceUnit.KILOMETERS)
    .addUnboundedTo(500)
    .addRange(500, 1000)
    .addUnboundedFrom(1000);
SearchResponse response = EsUtils.getSingleTransportClient()
    .prepareSearch("geo")
    .addAggregation(geoDistanceAgg)
    .execute().actionGet();
Range agg = response.getAggregations().get("agg");
for (Range.Bucket entry : agg.getBuckets()) {
     String key = entry.getKeyAsString();
     Number from = (Number) entry.getFrom();
     Number to = (Number) entry.getTo();
     long docCount = entry.getDocCount();
     System.out.println(key + "--" + docCount);
}
```

- IP Range Aggregation

```
IpRangeAggregationBuilder ipAgg = AggregationBuilders
     .ipRange("agg")
     .field("ip")
     .addUnboundedTo("100.0.0.5")
    .addUnboundedFrom("100.0.0.5 ");
 SearchResponse response = EsUtils.getSingleTransportClient()
     .prepareSearch("ip_test")
     .addAggregation(ipAgg)
     .execute().actionGet();
 Range agg = response.getAggregations().get("agg");
 for (Range.Bucket entry : agg.getBuckets()) {
       String key = entry.getKeyAsString();
       String fromAsString = entry.getFromAsString();
       String toAsString = entry.getToAsString();
       long docCount = entry.getDocCount();
       System.out.println(key+"---"+docCount);
 }
```

8.9　集群管理

和索引管理类似，集群管理通过创建 ClusterAdminClient 对象可以获取集群和索引的健康状态、集群状态。创建 ClusterAdminClient 对象的方法如代码清单 8-3 所示。

```
ClusterHealthResponse healths = client().admin().cluster()
        .prepareHealth().get();
String clusterName = healths.getClusterName();
int numberOfDataNodes = healths.getNumberOfDataNodes();
int numberOfNodes = healths.getNumberOfNodes();
for (ClusterIndexHealth health : healths.getIndices().values()) {
    String index = health.getIndex();
    int numberOfShards = health.getNumberOfShards();
    int numberOfReplicas = health.getNumberOfReplicas();
    ClusterHealthStatus status = health.getStatus();
}
```

8.10　本章小结

本章介绍了 Elasticsearch Java API 的使用方法，重点介绍了如何通过客户端对象进行文档的 CRUD、搜索、聚合等操作。

第9章 集群管理

本章学习要点：

- ✱ 集群规划
- ✱ 索引规划
- ✱ 分布式集群配置
- ✱ 查看集群健康状态
- ✱ 如何使用监控插件

9.1 集群规划

当我们计划搭建一个 Elasticsearch 集群的时候，面临的第一个问题是这个集群需要多少个节点？一个集群中有多少个节点受多种因素的影响，就数据量而言，如果数据量不大，那么几台机器就能满足需求；如果数据量非常庞大，需要几百台机器也是有可能的。一个集群中使用的硬件资源要根据业务的数据量大小来确定，如果条件允许，服务器性能和服务器数量当然是多多益善。

节点数确定以后会面临第二个问题：集群应该设置多少个 master 节点？在回答这个问题之前，我们先来了解一下什么是脑裂。所谓脑裂问题，就是同一个集群中的不同节点对于集群的状态有了不一样的理解，脑裂问题是分布式集群环境中必然会遇到的问题。

首先看一个有两个节点的 Elasticsearch 集群的简单情况。集群维护一个单个索引并有一个分片和一个副本。如图 9-1 所示，节点 1 在启动时被选举为主节点并保存主分片（0P），而节点 2 保存复制分片（0R）。

假设现在由于网络问题或其他原因，两个节点之间的通信发生中断，如图 9-2 所示。

这时两个节点都相信对方已经挂了，节点 1 不需要做什么，因为它本身就被选举为主节点，但是节点 2 会自动选举它自己为主节点，因为节点 2 和集群的主机点之间已经无法通信了，如图 9-3 所示。在 Elasticsearch 集群中是由主节点来决定将分片分配到从节点的，节点 2 保存的是副本分片，但它相信主节点不可用了，所以它会自动提升从节点为主节点。

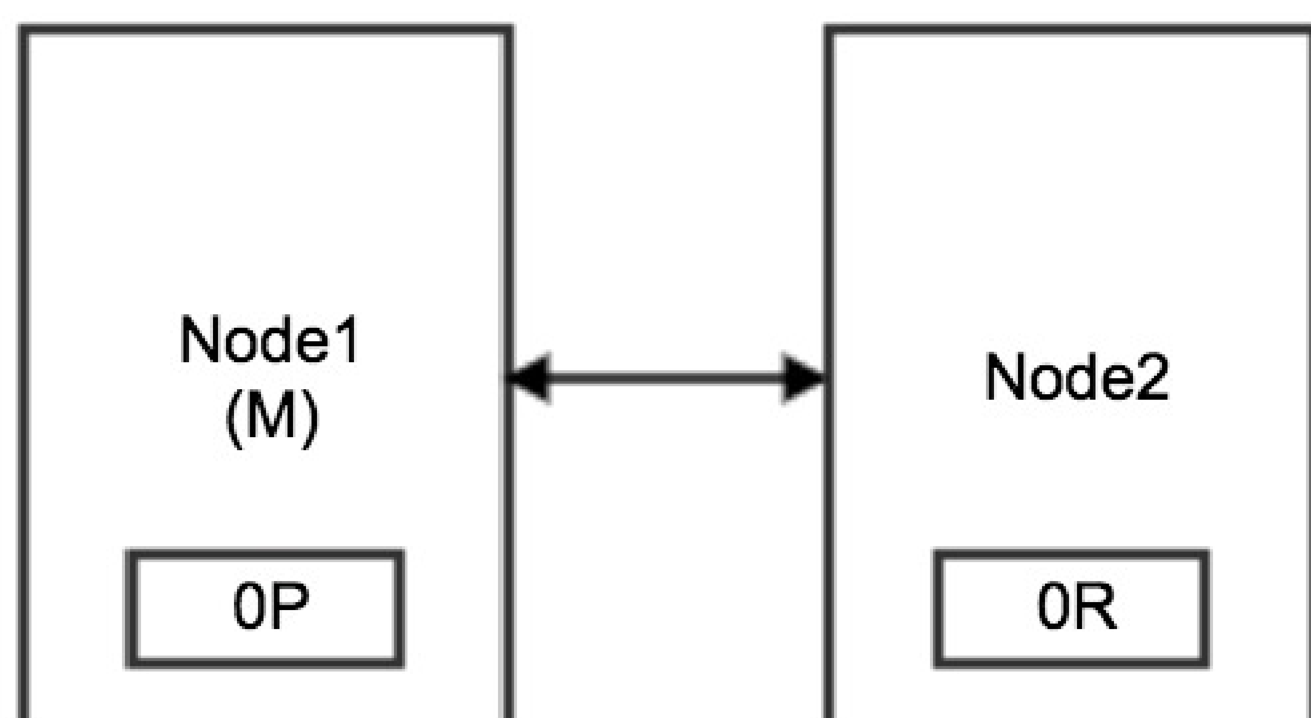


图 9-1　两个节点的 Elasticsearch 集群

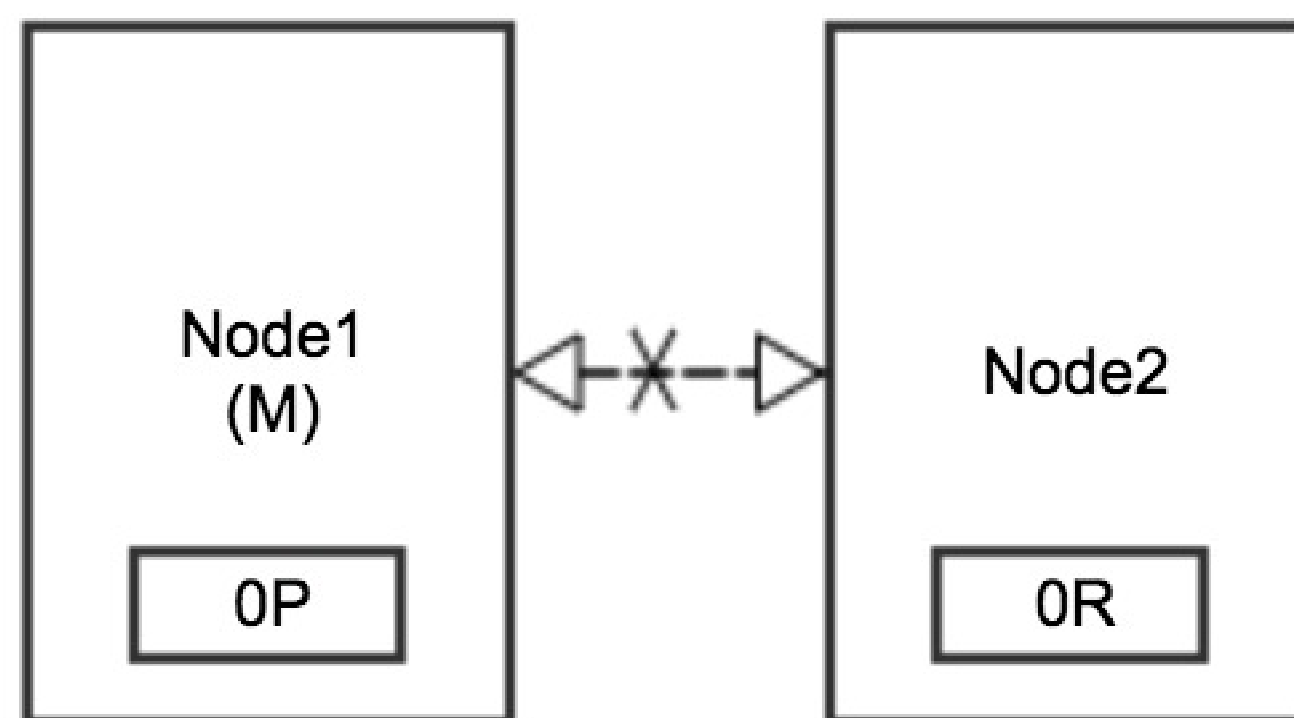


图 9-2　两个节点的 Elasticsearch 集群网络中断

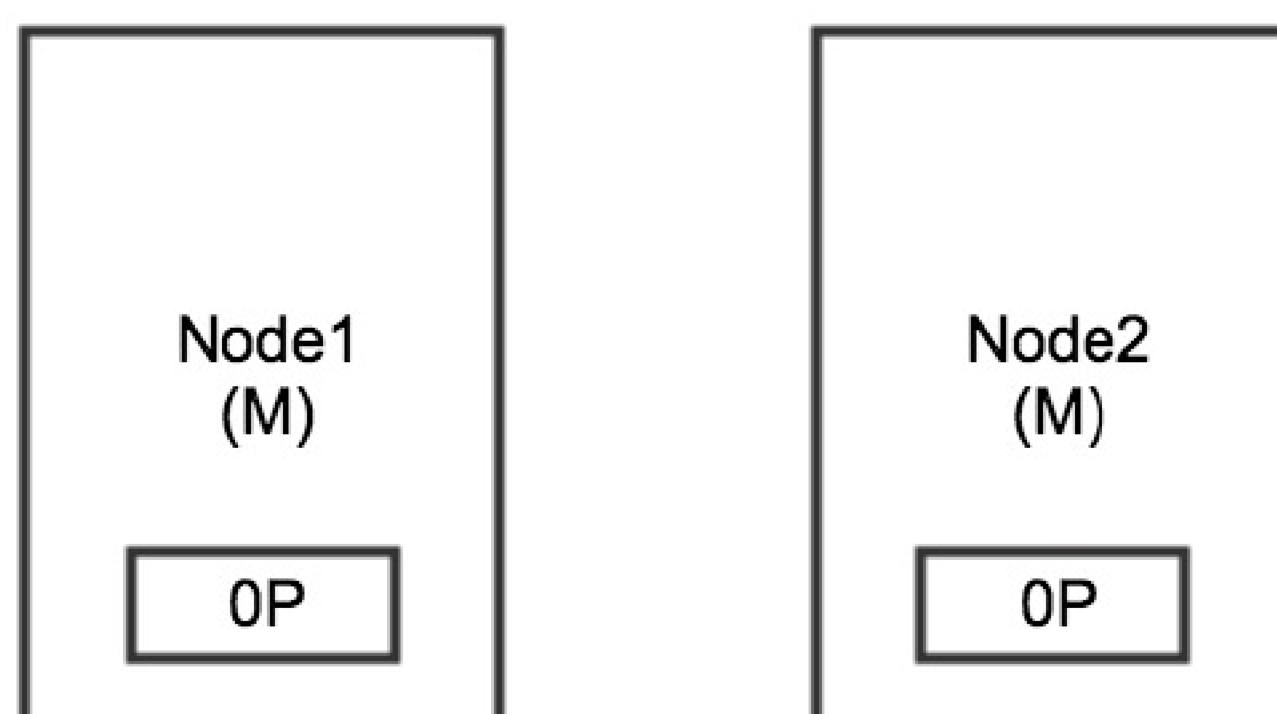


图 9-3　两个节点的 Elasticsearch 集群各自为 Master 节点

现在集群处于不一致的状态，发送到节点 1 上的索引请求不会将数据分配到节点 2，同时发送到节点 2 的请求也不会将数据分配到节点 1。在这种情况下，分片的两份数据分开了，如果不做一个全量的重索引，就很难对它们进行重排序。在更坏的情况下，一个对集群无感知的索引客户端（例如使用 REST 接口的），这个问题非常透明难以发现，请求仍然会成功完成。问题只有在搜索数据时才会被隐约发现：取决于搜索请求命中了哪个节点，结果都会不同。

要避免脑裂的发生，可以在 Elasticsearch 的配置文件（config 目录中的 elasticsearch.yml）中做一些避免脑裂的配置。一个常用的参数是 discovery.zen.minimum_master_nodes，这个参数决定了主节点选择过程中最少需要有多少个 master 节点，默认配置是 1。一个基本的原则是这里需要设置成 N/2+1，N 是集群中节点的数量。例如在一个 3 节点的集群中，minimum_master_nodes 应该被设为 2。

我们再来看之前两个节点的情况，如果我们把 discovery.zen.minimum_master_nodes 设置成 2，当两个节点的通信失败时，节点 1 会失去它的主状态，同时节点 2 也不会被选举为主节点。

避免脑裂的另一个参数是 discovery.zen.ping.timeout，它的默认值是 3 秒，并且它用来决定节点之间网络通信的等待时间。如果网络环境较差，可以将这个值调的大一点。这个参数不仅适用于高网络延迟的情况，还能在一个节点超载响应慢时起作用。

2 节点集群中把 minimum_master_nodes 参数设成 2 可以避免脑裂的发生，但是在这种情况下如果一个节点挂了，整个集群就都挂了。如果你刚开始使用 Elasticsearch，建议配置一个 3 节点集群，设置 minimum_master_nodes 为 2，这样可以减少脑裂的可能性并保持高可用的优点，即使一个节点失效，但集群还是可以正常运行的。

9.2　索引规划

Elasticsearch 集群搭建完成以后创建索引是开始搜索之旅的第一步，默认情况下一个索引的分片数是 5，副本数是 1。我们知道分片是把一个大的索引分成多份放到不同的节点上来加速查询效率，如图 9-4 所示，当用户发送一个查询请求以后，Elasticsearch 会把请求转发到不同的节点上，分别到各个分片上进行搜索，然后把各个分片的搜索结果合并，最后返回搜索结果。分片过少，数据量很大时，索引文件也会很大，不能发挥分布式搜索的优点；分片数过多，在分发查询请求、合并搜索结果的时候会浪费时间。那么一个索引的分片为 5 是否总能满足需求？创建索引时应该如何设置分片数？

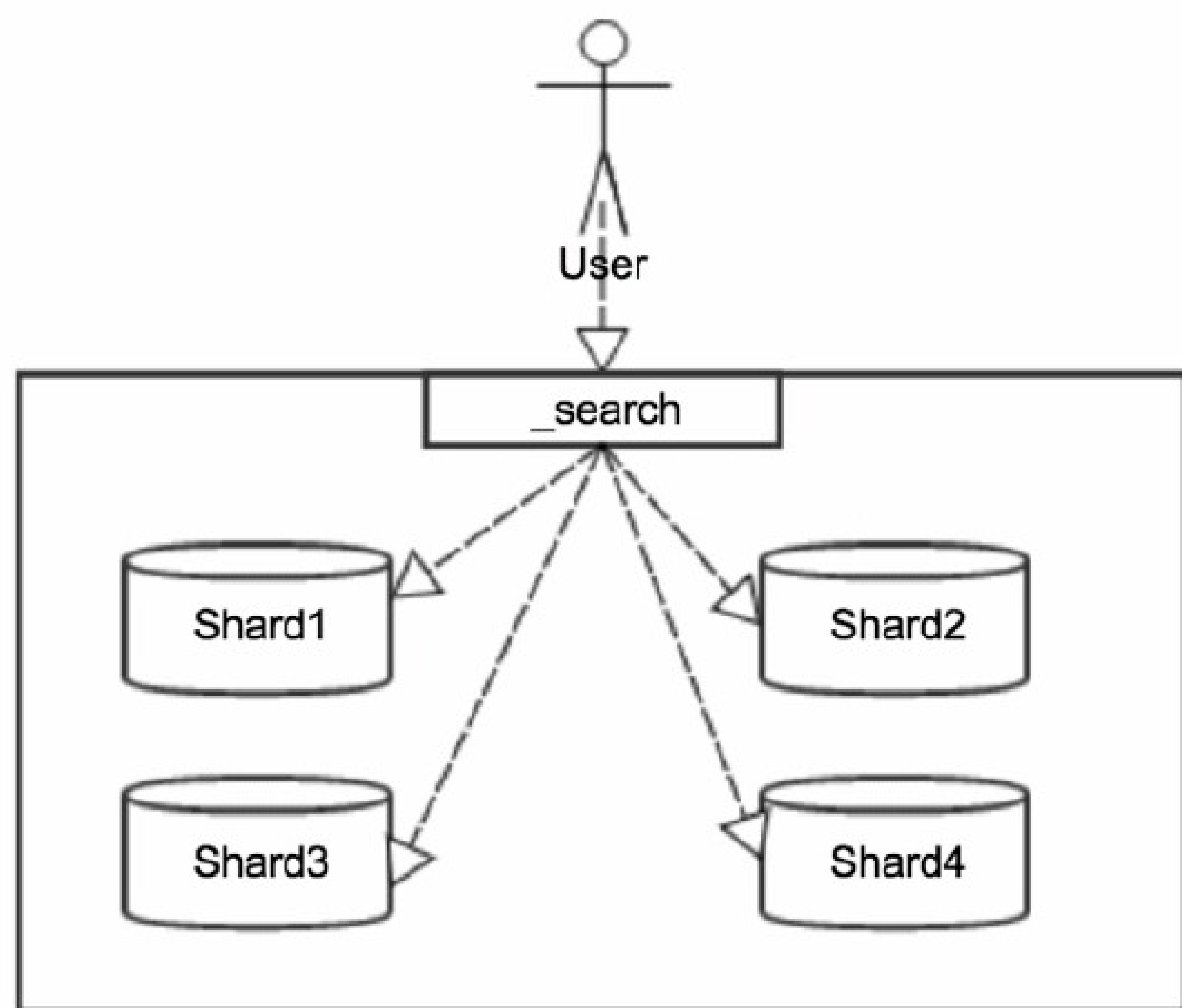


图 9-4　Elasticsearch 搜索分片过程

分片数量的确定从根本上来讲是看查询的效率，有些情况下响应时间要求是毫秒级，有些情况下要求是秒级，所以分片的数量以能够最大化查询效率为原则。查询的响应时间受多个变量的影响，挑出一些重要的总结如下。

- 服务器的性能：服务器的内存大小、CPU 性能等参数对查询效率有很大的影响，同样的数据分成相同多个分片，性能好的服务器响应时间更快。
- 硬盘：使用普通硬盘和 SSD 高度硬盘在性能上有着很大的差别，同样的数据，在普通硬盘上的查询时间不能满足系统对响应时间的要求，也许数据放到 SSD 上就能满足。
- 文档结构的复杂度：复杂结构的文档比结构简单的文档需要消耗更多的资源，查询时间也会延长，文档结构的复杂程度会影响搜索效率。
- 查询语句的复杂程度：复杂查询，尤其是嵌套搜索、聚合分析会比简单查询更加耗时。

Elasticsearch 官网提供了单机性能测试的方法，通过测试一台服务器上的一个分片估算 N 台服务器的性能，一旦获取单个分片的性能，再考虑整个文档的大小以及预期的增长等因素就可以确定 N 台服务器下的分片数。测试步骤如下：

步骤 01 创建一个单节点组成的服务器，使用生产环境下相同的硬件配置。

步骤 02 创建一个索引，使用生产环节下要使用的设置和分词器，索引只设置一个主分片，不设置副本。

步骤 03 索引真实（或者尽可能真实）的文档到索引中。

步骤 04 执行尽可能接近真实的查询和聚合。

在做性能测试之前，应尽可能地先优化 Elasticsearch，比如说检查是否使用了低效率的查询，检查服务器内存是否足够大，检查 swap 交换区是否足够大。

9.3　分布式集群

Elasticsearch 集群中的节点一般有 3 种角色，在搭建完全分布式集群以前需要在配置文件中指定节点的角色，简介如下。

- master 节点：master 节点主要负责元数据的处理，比如索引的新增、删除、分片分配等，每当元数据有更新时，master 节点负责同步到其他节点上。
- data 节点：data 节点上保存了数据分片。它负责数据相关操作，比如分片的增删改查以及搜索和整合操作。
- client 节点：client 节点起到路由请求的作用，实际上可以看作负载均衡器，适用于高并发访问的业务场景。

准备 3 台 Linux 虚拟机，虚拟地址分别为 10.90.4.7、10.90.4.8 和 10.90.4.9，在这 3 台虚拟机上分别安装好 Elasticsearch 5.4.0，集群名称设置为 ucas，节点名依次为 node-07、node-08、node-09。为了避免脑裂，3 台机器的集群 master 节点应为 2 个，选取 10.90.4.7 和 10.90.4.8 这两个节点上的 Elasticsearch 作为 master 节点，10.90.4.9 的节点只作为 client 节点（如果没有处理高并发访问的需求，client 节点可以不添加，这里只做演示）。集群的架构图如图 9-5 所示。

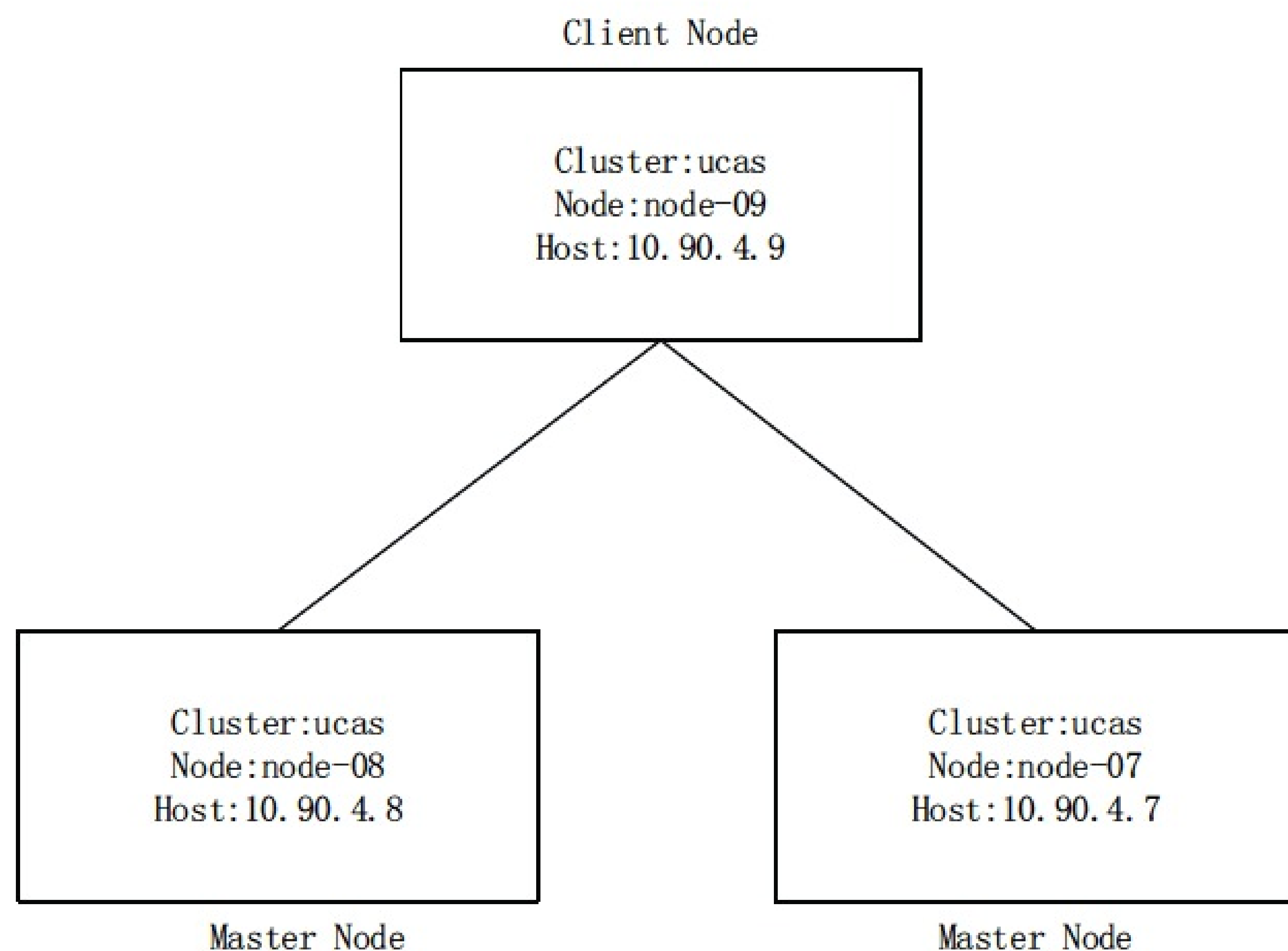


图 9-5　Elasticsearch 集群架构图

各节点 config/elasticsearch.yml 中的配置如下：

- 10.90.4.7 服务器上的配置

```
cluster.name: ucas
node.name: node-07
network.host: 10.90.4.7
http.port: 9200
http.cors.enabled: true
http.cors.allow-origin: "*"
node.master: true
node.data: true
discovery.zen.ping.unicast.hosts: ["10.90.4.7","10.90.4.8"]
```

- 10.90.4.8 服务器上的配置

```
cluster.name: ucas
node.name: node-08
network.host: 10.90.4.8
http.port: 9200
http.cors.enabled: true
http.cors.allow-origin: "*"
node.master: true
node.data: true
discovery.zen.ping.unicast.hosts: ["10.90.4.7","10.90.4.8"]
```

- 10.90.4.9 服务器上的配置

```
cluster.name: ucas
node.name: node-09
network.host: 10.90.4.9
http.port: 9200
http.cors.enabled: true
http.cors.allow-origin: "*"
node.master: false
node.data: false
discovery.zen.ping.unicast.hosts: ["10.90.4.7","10.90.4.8"]
```

如图 9-6 所示，配置完成以后分别启动各个虚拟机中的 Elasticsearch，通过 Head 插件访问可以看到一个 3 节点集群。

集群搭建好以后有一项很重要的工作，就是对集群的使用情况进行监控，对于一个多节点集群，由于网络连接问题，出现宕机、脑裂等异常情况都是有可能发生的。Elasticsearch 提供了 Cat API 和 Cluster API，可以方便地获取集群的健康情况、集群状态、节点状态、索引统计等信息。

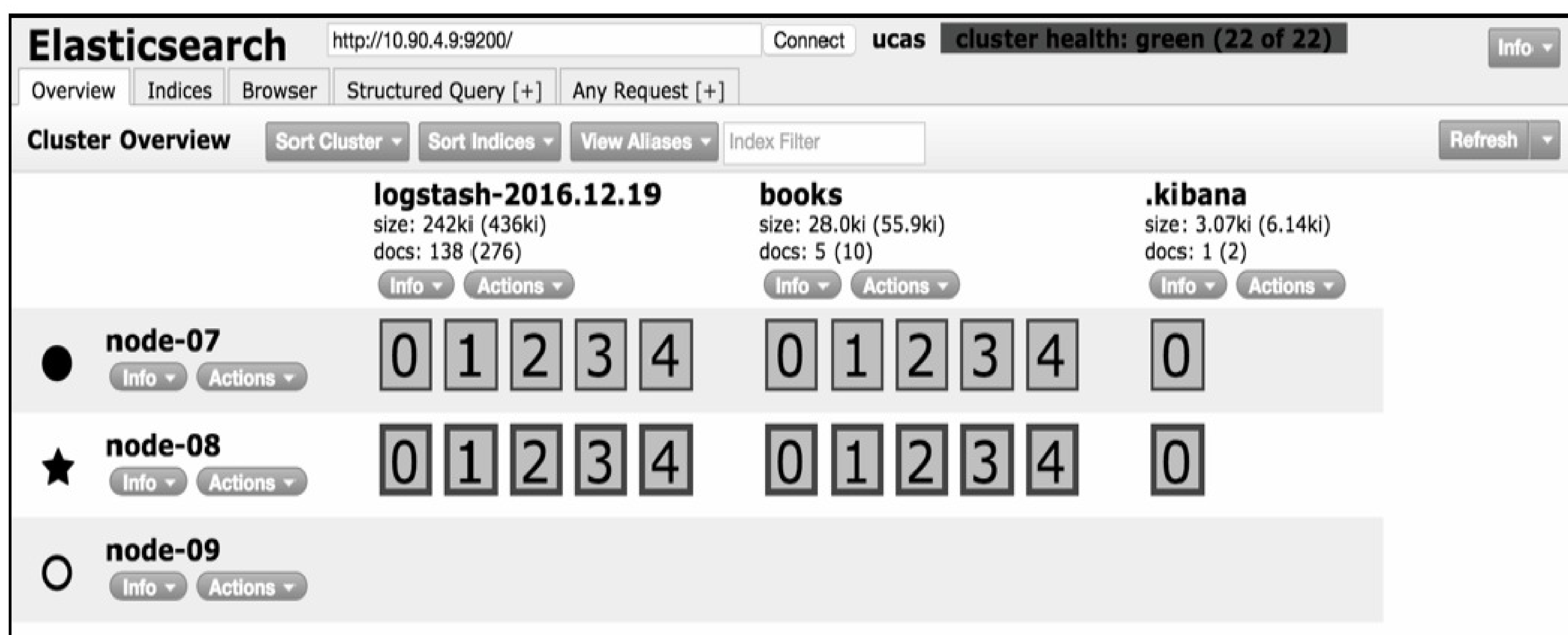


图 9-6　Head 插件中访问 Elasticsearch 集群

9.4　Cat API

9.4.1　cat aliases

cat aliases 命令用于显示索引的别名，也包括过滤器和路由信息。命令如下：

```
GET /_cat/aliases?v
```

可能的响应信息如下：

```
alias  index filter routing.index routing.search
alias1 test1    -         -           -
alias2 test1    *         -           -
alias3 test1    -         1           1
alias4 test1    -         2           1,2
```

9.4.2　cat allocation

cat allocation 命令可以查看每个节点分片的分配数量以及它们所使用的硬盘空间大小。命令如下：

```
GET /_cat/allocation?v
```

可能的响应信息如下：

```
shards  disk.indices disk.used disk.avail disk.total disk.percent host ip    node
85    32.4mb     147.3gb  52.6gb  199.9gb   73  172.31.44.9  172.31.44.9 ovSPGIe
```

9.4.3　cat count

cat count 命令可以快速查询整个集群或者单个索引的文档数量。命令如下：

```
GET /_cat/count?v
```

响应信息如下：

```
epoch       timestamp  count
1505666033  00:33:53   5616
```

查看一个索引的文档数量：

```
GET /_cat/count/books?v
```

9.4.4　cat fielddata

cat fielddata 命令用于查看当前集群中每个数据节点上被 fielddata 所使用的堆内存大小。命令如下：

```
GET /_cat/fielddata?v
```

响应信息如下：

```
id               host         ip           node      field             size
ovSPGIe       172.31.44.9  172.31.44.9  ovSPGIe  _parent#branch     704b
ovSPGIe       172.31.44.9  172.31.44.9  ovSPGIe  _parent#question  376b
ovSPGIe       172.31.44.9  172.31.44.9  ovSPGIe  _parent            1kb
```

9.4.5　cat health

cat health 命令用于显示集群的健康信息。命令如下：

```
GET /_cat/health?v
```

9.4.6　cat indices

cat indices 命令可以查看索引信息，包括索引健康状态、索引开关状态、分片数、副本数、文档数量、标记为删除的文档数量、占用的存储空间等信息。命令如下：

```
GET /_cat/indices/?v
```

上面的命令会返回集群中所有索引的信息，也可以查看一个索引的信息：

```
GET /_cat/indices/books?v
```

响应信息如下：

```
    health status index  uuid  pri  rep docs.count docs.deleted store.size
pri.store.size
    green    open  books  yYOp  5   0          6       0      27.4kb      27.4kb
```

9.4.7　cat master

cat master 命令可以显示 master 节点的节点 ID、绑定的 IP 和节点名。命令如下：

```
GET /_cat/master?v
```

响应信息如下：

```
id                    host          ip            node
ovSPGIe              172.31.44.9   172.31.44.9    ovSPGIe
```

9.4.8　cat nodeattrs

cat nodeattrs 命令可以显示指定节点的属性信息。命令如下：

```
GET /_cat/nodeattrs?v
```

响应信息如下：

```
node     host     ip          attr   value
DKDM97B  epsilon  192.168.1.8  rack   rack314
DKDM97B  epsilon  192.168.1.8  azone  us-east-1
```

9.4.9　cat nodes

cat nodes 命令可以查看集群拓扑结构。命令如下：

```
GET /_cat/nodes/?v
```

响应信息如下：

```
ip heap.percent ram.percent cpu load_1m load_5m load_15m node.role master name
172.31.44.9  13          76     4              mdi     *            ovSPGIe
```

9.4.10　cat pending tasks

cat pending tasks 命令用于查看正在执行的任务列表。命令如下：

```
GET /_cat/pending_tasks?v
```

响应信息如下：

```
insertOrder  timeInQueue  priority      source
 1685       855ms       HIGH     update-mapping [foo][t]
 1686       843ms       HIGH     update-mapping [foo][t]
 1693       753ms       HIGH     refresh-mapping [foo][[t]]
```

9.4.11　cat plugins

cat plugins 命令用于查看每一个节点所运行插件的信息。命令如下：

```
GET /_cat/plugins?v
```

响应信息如下：

```
name     component    version
ovSPGIe  analysis-ik    5.4.0
ovSPGIe  mapper-size    5.4.0
```

9.4.12 cat recovery

cat recovery 命令是一个索引分片恢复的视图，包括恢复中的和先前已完成的。命令如下：

```
GET /_cat/recovery/books?v
```

9.4.13 cat repositories

cat repositories 命令用于展示集群中注册的快照库。命令如下：

```
GET /_cat/repositories?v
```

响应信息如下：

```
id      type
repo1   fs
repo2   s3
```

9.4.14 cat thread pool

thread_pool 命令用于展示集群中每一个节点线程池的统计信息。默认情况下返回所有线程池的 active、queue 和 rejected 的统计信息。命令如下：

```
GET /_cat/thread_pool?v
```

响应信息如下：

```
node_name name                active queue rejected
ovSPGIe   bulk                    0     0        0
ovSPGIe   fetch_shard_started     0     0        0
ovSPGIe   fetch_shard_store       0     0        0
ovSPGIe   flush                   0     0        0
ovSPGIe   force_merge             0     0        0
ovSPGIe   generic                 0     0        0
ovSPGIe   get                     0     0        0
ovSPGIe   index                   0     0        0
ovSPGIe   listener                0     0        0
ovSPGIe   management              1     0        0
ovSPGIe   refresh                 0     0        0
ovSPGIe   search                  0     0        0
ovSPGIe   snapshot                0     0        0
ovSPGIe   warmer                  0     0        0
```

9.4.15 cat shards

cat shards 命令用于查看节点包含的分片信息，包括一个分片是主分片还是一个副本分片、文档的数量、硬盘上占用的字节数、节点所在的位置等信息。命令如下：

```
GET /_cat/shards/books?v
```

响应信息如下：

```
index shard prirep   state       docs  store    ip              node
books 3     p       STARTED       2    10.4kb  172.31.44.9     ovSPGIe
books 1     p       STARTED       1    5kb     172.31.44.9     ovSPGIe
books 4     p       STARTED       1    5.4kb   172.31.44.9     ovSPGIe
books 2     p       STARTED       2    6.3kb   172.31.44.9     ovSPGIe
books 0     p       STARTED       0    159b    172.31.44.9     ovSPGIe
```

9.4.16　cat segments

cat segments 命令用于查看索引的段信息，命令如下：

```
GET /_cat/segments/books?v
```

响应信息如下：

```
index shard prirep  ip segment generation docs.count docs.deleted [...]
books  1      p   172.31.44.9     _0         0          1        0
books  2      p   172.31.44.9     _0         0          2        0
books  3      p   172.31.44.9      _0        0          1        0
[...] size    size.memory  committed  searchable  version  compound
4.8kb    2592             true       true       6.5.0   true
 6.1kb    2976             true       true       6.5.0   true
 5.1kb    2592             true       true       6.5.0   true
```

9.4.17　cat templates

cat templates 命令用于查看集群中的模板，命令如下：

```
GET /_cat/templates?v
```

响应信息如下：

```
name       template order version
template_1   te*      0
```

9.5　Cluster API

9.5.1　Cluster Health

利用 Elasticsearch 的集群健康 API 可以查看当前集群的健康信息，命令如下:

```
GET  _cluster/health
```

返回结果如下：

```
{
  "cluster_name": "elasticsearch",
  "status": "green",
```

```
  "timed_out": false,
  "number_of_nodes": 1,
  "number_of_data_nodes": 1,
  "active_primary_shards": 85,
  "active_shards": 85,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 0,
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "number_of_in_flight_fetch": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 100
}
```

返回结果中各个属性的含义解释如下。

- cluster_name：集群名称。
- status：集群的健康状态，颜色的含义如表 9-1 所示。
- timed_out：是否超时。
- number_of_nodes：节点数，包括 master 节点和 data 节点。
- number_of_data_nodes：data 节点数。
- active_primary_shards：活动的主分片。
- active_shards：所有活动的分片数，包括主分片和副本。
- relocating_shards：正在发生迁移的分片。
- initializing_shards：正在初始化的分片。
- unassigned_shards：没有被分配的分片。
- delayed_unassigned_shards：延迟未被分配的分片。
- number_of_pending_tasks：master 节点任务队列中的任务数。
- number_of_in_flight_fetch：正在进行迁移的分片数量。
- task_max_waiting_in_queue_millis：队列中任务的最大等待时间。
- active_shards_percent_as_number：活动分片的百分比。

上面的命令用于获取整个集群的健康信息，也可以增加参数（索引名称），获取一个或多个索引的健康信息，命令如下：

```
GET /_cluster/health/books
```

表9-1　集群健康颜色的含义

状态	意义
green	所有主分片和从分片都可用
yellow	所有主分片可用，但存在不可用的从分片
red	存在不可用的主分片

加上 level 参数，获取分片级别的健康信息：

```
GET /_cluster/health/books?level=shards
```

9.5.2　Cluster State

Cluster State（集群状态）API 可以对整个集群的信息进行一个全面的了解，包括集群信息、集群中每个节点的信息、元数据、路由表等。查看集群状态的命令如下：

```
GET /_cluster/state
```

该命令返回的信息多达几百行，这里折叠处理，截图如图 9-7 所示。查看集群状态时也可以添加以下参数进行过滤处理，只返回部分信息。

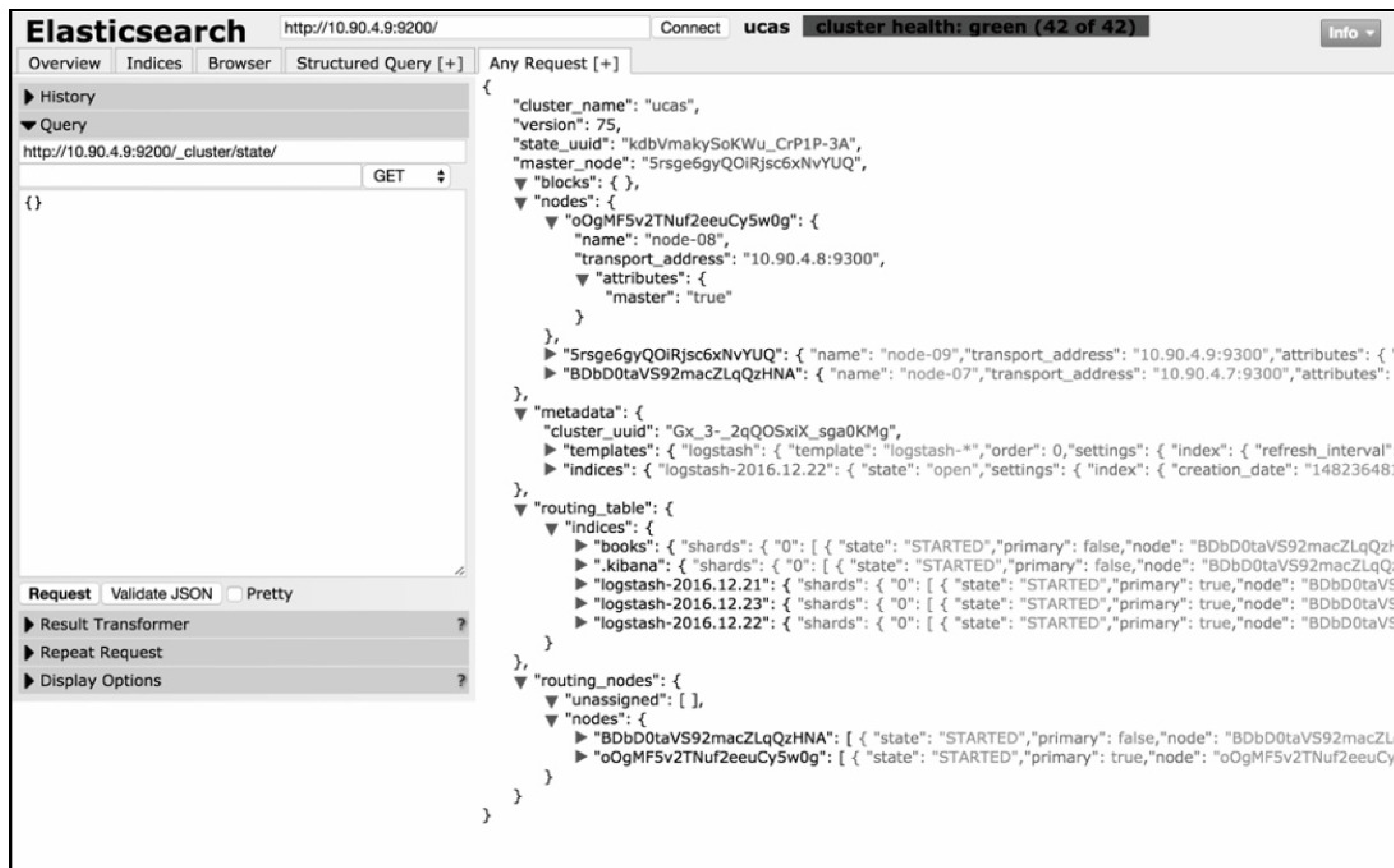


图 9-7　Head 插件中查看集群状态

- version：返回集群状态版本信息。
- master_node：只返回 master 节点的状态信息。
- nodes：返回集群中的节点的配置信息，主要包括节点名称、IP、是否是 master 节点。
- routing_table：返回每个节点的路由信息。
- metadata：返回元数据信息，包括每个索引的 mapping、setting 等信息。
- blocks：返回集群中的块数据信息。

下面给出几个例子。

（1）只返回集群的版本信息：GET /_cluster/state/version。

（2）返回集群中节点的配置信息：GET /_cluster/state/nodes。

（3）返回 books、books2 这两个索引的 metadata 和 routing_table：

```
GET /_cluster/state/metadata,routing_table/books,books2
```

9.5.3　Cluster Stats

Cluster Stats（集群统计）API 用于从集群中获取各种统计数据。该 API 的返回信息主要有两部分，一部分是索引层面，包含分片数、存储大小、内存使用情况等指标，另一部分是节点层面，包含节点数量、节点角色、操作系统、jvm 版本、内存、CPU、安装的插件等指标。查看集群统计信息的命令如下，返回结果如图 9-8 所示。

```
GET /_cluster/stats
```



图 9-8　Head 插件中查看集群统计信息

9.5.4　Pending Cluster Tasks

Pending Cluster Tasks API 用于返回一个正在添加到更新集群状态的任务列表。集群中的变化通常是很快的，通常这个操作会返回一个空的列表。

```
GET /_cluster/pending_tasks
```

9.5.5　Cluster Reroute

reroute 命令可以明确地执行集群重新路由分配命令。例如，把一个分片从一个节点移动到另一个节点，把未分配的分片移动到一个指定的节点。例子如下：

```
POST /_cluster/reroute
{
  "commands": [
```

```
    {
      "move": {
        "index": "test",
        "shard": 0,
        "from_node": "node1",
        "to_node": "node2"
      }
    },
    {
      "allocate_replica": {
        "index": "test",
        "shard": 1,
        "node": "node3"
      }
    }
  ]
}
```

9.5.6 Cluster Update Settings

Update Settings 命令可以更新集群中的配置，如果是永久配置，就需要重启集群；如果是瞬时配置，就不需要重启集群。例如，更新最小 master 节点数：

```
PUT /_cluster/settings
{
  "persistent": {
    "discovery.zen.minimum_master_nodes": 1
  }
}
```

9.5.7 Nodes Stats

Cluster Nodes Stats(集群节点统计信息)API 可以获取集群中一个或者多个节点的统计信息。获取集群中所有节点的统计信息的命令如下：

```
GET /_nodes/stats
```

获取 nodeId1 和 nodeId2 节点的统计信息的命令如下：

```
GET /_nodes/nodeId1,nodeId2/stats
```

9.5.8 Nodes Info

Cluster Nodes Info API 可以获取集群中一个或多个节点的信息，包括设置、操作系统、虚拟机、线程池等信息。命令如下：

```
GET /_nodes
```

获取指定节点的信息：

```
GET /_nodes/nodeId1,nodeId2
```

也可以添加参数（settings、os、process、jvm、thread_pool、transport、http、plugins、ingest 和 indices）返回指定信息。例如，查看节点的 os 和 jvm 信息，命令如下：

```
GET /_nodes/os,jvm
```

9.5.9 Task Management API

Task Management API 可用于获取 Elasticsearch 集群中一个或多个节点正在执行中的任务信息。命令如下：

```
GET /_tasks
```

9.5.10 Cluster Allocation Explain API

Cluster Allocation Explain API 用于解释分片没有被分配的原因。例如，查看 my-index 索引中的第 0 号分片，命令如下：

```
GET /_cluster/allocation/explain
{
  "index": "my-index",
  "shard": 0,
  "primary": true
}
```

9.6 监控插件

Bigdesk 是 Elasticsearch 的一个集群监控工具，可以通过它来查看 ES 集群的各种状态，如 CPU、内存使用情况、JVM 信息、索引信息、搜索情况、HTTP 连接数、磁盘系统信息等。Bigdesk 托管在 GitHub 上，项目地址为 https://github.com/hlstudio/bigdesk。安装 Bigdesk 插件的方法如下。

首先执行 git clone 命令，下载 bigdesk 源码：https://github.com/hlstudio/bigdesk.git。

然后在浏览器中打开 bigdesk-master_site 目录下的 index.html，在 ES node REST endpoint 输入框中输入 Elasticsearch 的连接地址和端口即可。

- Summary：从左到右依次为内存和 CPU 的使用率、Heap 内存使用情况、GC 次数和时间、索引段的统计信息，如图 9-9 所示。
- Indices：查看索引数据和查询情况。上面 4 个依次代表每秒的搜索请求次数、每秒的搜索次数、每秒的索引请求次数和每秒的索引次数，下面 4 个依次代表缓冲区大小、缓存失效个数、每秒 get 请求数量和每秒的 get 次数，如图 9-10 所示。

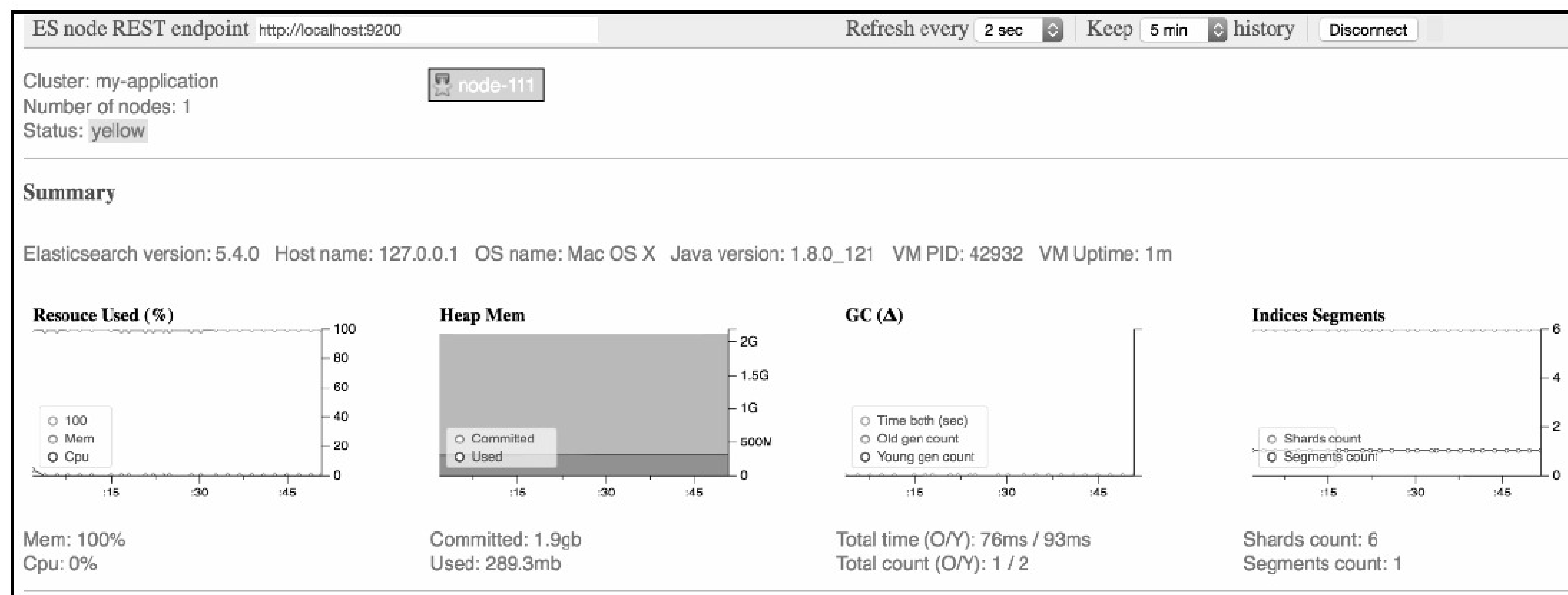


图 9-9 Summary 信息

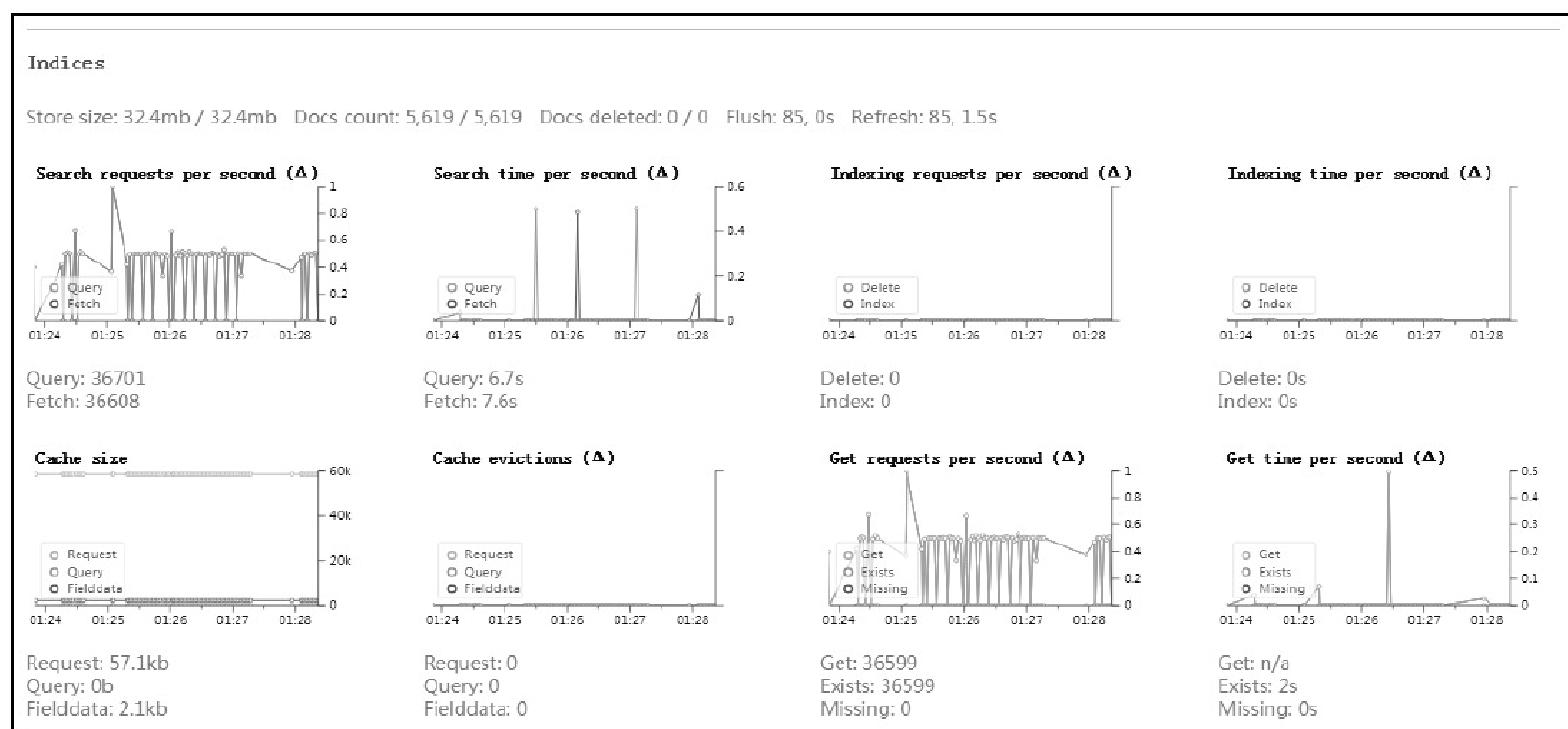


图 9-10 Indices 信息

- Thread Pools：从左到右依次为 Search、Index、Bulk、Refresh 的线程统计情况，有队列中的请求数、峰值次数、统计次数 3 个指标，如图 9-11 所示。

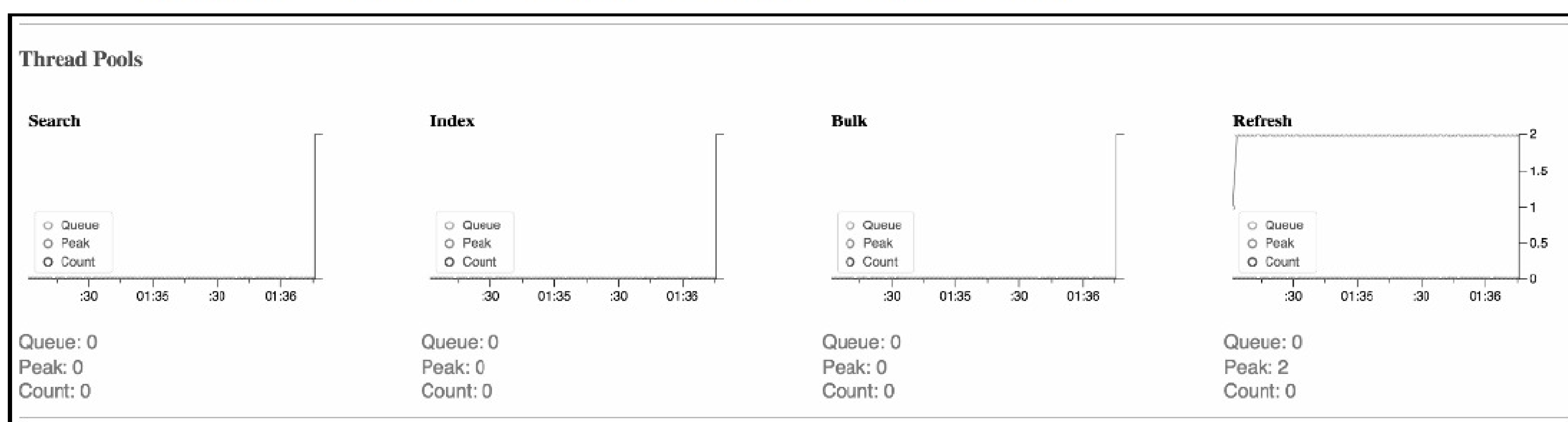


图 9-11 Thread Pools 信息

- OS & JVM & Process & Transport：服务器信息、处理器信息、线程数、内存使用情况（底部蓝色为已使用的内存）、HTTP 和 TCP 连接数，如图 9-12 所示。

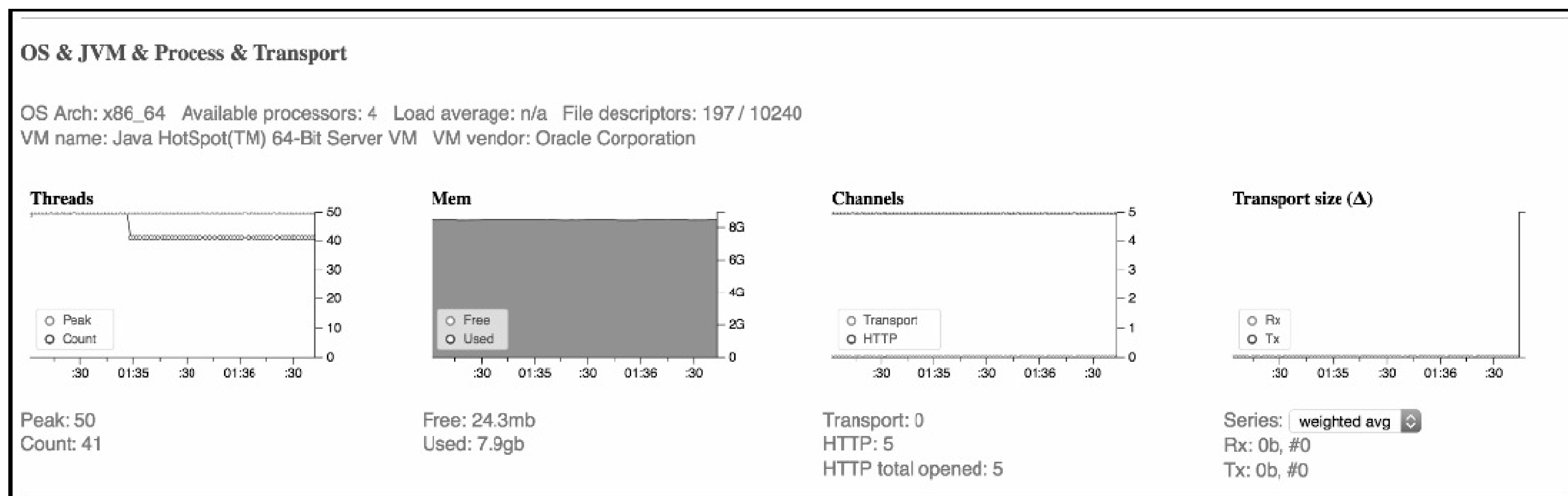


图 9-12　OS & JVM & Process & Transport 信息

- File system：查看文件系统类型、挂载路径、总的磁盘大小和可用的磁盘大小，path 表示 Elasticsearch 的安装路径，如图 9-13 所示。

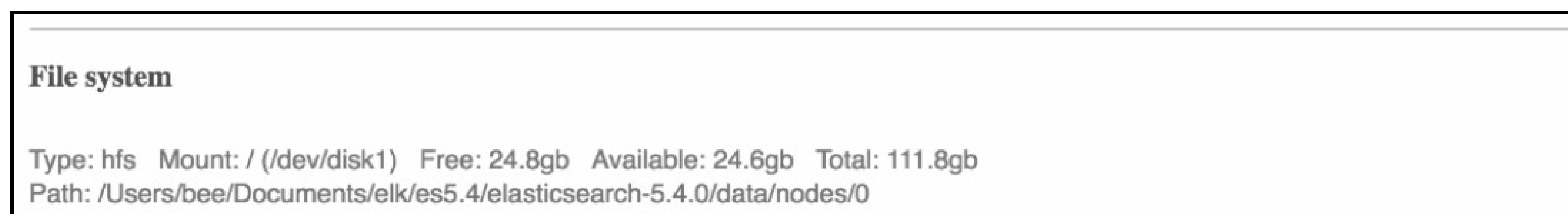


图 9-13　File system 信息

9.7　本章小结

本章介绍了 Elasticsearch 集群管理的相关知识点，包括脑裂问题、集群规划、索引规划、分布式集群的搭建方法以及如何查看集群的监控信息。

第10章 新闻搜索项目实战

本章学习要点：

- ✱ Elasticsearch 整合 MySQL 需求分析
- ✱ 索引 MySQL 数据到 Elasticsearch
- ✱ 新闻搜索框和搜索结果页设计
- ✱ 搜索结果展示
- ✱ 关键字高亮
- ✱ 搜索结果分页

10.1 需求分析

全文检索的一个典型应用是对关系型数据库中的数据建立索引进行搜索。这一章通过新闻搜索的案例总结前几章的知识点，通过项目实战来学习 Elasticsearch 与 MySQL 的整合应用，新闻搜索的架构设计如图 10-1 所示，需要实现的需求如下：

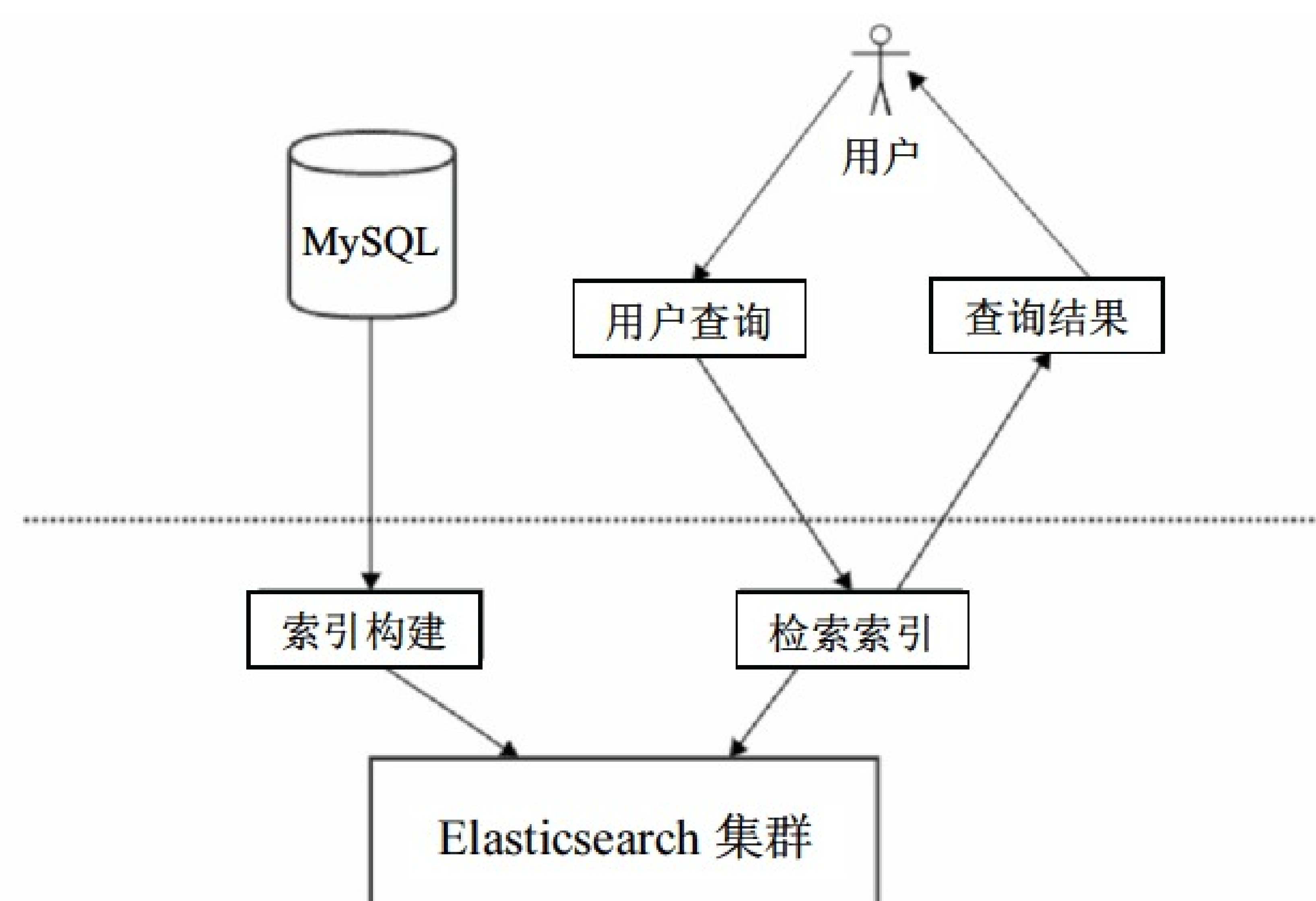


图 10-1　Elasticsearch 新闻搜索系统架构图

（1）能够对新闻标题进行关键词检索。
（2）能够对新闻内容进行关键词检索。
（3）能够实现关键字的高亮。
（4）能够实现搜索结果分页显示。

数据导入完成以后，创建一个 Maven 工程，在 pom.xml 中添加 servlet、mysql-connector-java、log4j2 和 Elasticsearch 的依赖：

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>6.0.6</version>
</dependency>

<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>transport</artifactId>
  <version>5.4.0</version>
</dependency>

<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.8.2</version>
</dependency>

<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.8.2</version>
</dependency>
```

10.2　数据准备

为了方便学习，这里准备了几千条体育新闻作为测试数据，数据保存在 news.sql 文件中。导入数据到 MySQL 的步骤如下：

步骤 01 命令行登录 MySQL。

执行登录命令：mysql　-uroot -p，按回车键后输入 MySQL 密码，即可进入 MySQL 的命令行界面。

步骤 02 新建数据库，库名为 News。

执行命令：create database News。

步骤 03 使用 News 数据库。

执行命令：use News。

步骤 04 执行 SQL 脚本，导入数据。

执行命令：source /your/path/news.sql，路径为 news.sql 的绝对路径。

步骤 05 查看数据。

新闻主要有 id、标题、关键词、新闻内容、新闻原始 url、新闻评论数量、新闻来源、新闻发布时间这 8 个字段，表结构如下：

```
mysql> DESC news;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| id       | bigint(4)    | NO   | PRI | 0       |       |
| title    | varchar(100) | NO   |     | NULL    |       |
| key_word | varchar(50)  | YES  |     | NULL    |       |
| content  | text         | YES  |     | NULL    |       |
| url      | varchar(200) | YES  |     | NULL    |       |
| reply    | int(4)       | YES  |     | NULL    |       |
| source   | varchar(50)  | YES  |     | NULL    |       |
| postdate | datetime     | NO   |     | NULL    |       |
+----------+--------------+------+-----+---------+-------+
```

其中一条新闻的内容如下。

```
id: 1
title: 里皮让国足选帅从未如此众望所归再踢不好赖谁
key_word: 里皮,世预赛,国足
content: 新浪体育讯 千呼万唤始出来！随着里皮正式与中国足协签约，成为
         中国男足新一任主帅，国足正式进入里皮时代...
url:http://sports.sina.com.cn/china/national/2016-10-22/doc-37.shtml
reply: 9784
source: sina
time: 2016-10-22 05:55:00
```

导入完成以后可以使用 SQL 命令查看新闻数据的具体内容，也可以在 Navicat 中导入数据、查看数据，如图 10-2 所示。

图 10-2　Navicat 中查看新闻数据

10.3　数据导入

把 MySQL 中的数据导入 Elasticsearch 的步骤为创建索引、设置索引的映射、通过 JDBC 导入数据。首先创建单例模式的 TransportClient 对象：

```
public static final String CLUSTER_NAME = "elasticsearch";
public static final String HOST_IP = "172.31.44.9";
public static final int TCP_PORT = 9300;
private static volatile TransportClient client;

static Settings settings = Settings.builder()
    .put("cluster.name", CLUSTER_NAME)
    .build();
public static TransportClient getSingleClient() {
    if (client == null) {
        synchronized (TransportClient.class) {
          if (client == null) {
           try {
             client = new PreBuiltTransportClient(settings)
                 .addTransportAddress(new InetSocketTransportAddress(
                 netAddress.getByName(HOST_IP), TCP_PORT));
            } catch (UnknownHostException e) {
              e.printStackTrace();
            }
```

```
            }
          }
        }
        return client;
    }
```

封装一个创建索引的方法，索引名、分片数和副本数作为参数传入：

```
    public static IndicesAdminClient getAdminClient() {
        return getSingleClient().admin().indices();
    }

    public static boolean createIndex(String indexName, int shards,
        int replicas) {
        Settings settings = Settings.builder()
                .put("index.number_of_shards", shards)
                .put("index.number_of_replicas", replicas)
                .build();
        CreateIndexResponse createIndexResponse = getAdminClient()
                .prepareCreate(indexName.toLowerCase())
                .setSettings(settings)
                .execute().actionGet();
        boolean isIndexCreated = createIndexResponse.isAcknowledged();
        if (isIndexCreated) {
            System.out.println("索引" + indexName + "创建成功");
        } else {
            System.out.println("索引" + indexName + "创建失败");
        }
        return isIndexCreated;
    }
```

再封装一个设置映射的方法：

```
    public static boolean setMapping(String indexName, String typeName, String
         mapping) {
        getAdminClient().preparePutMapping(indexName)
                .setType(typeName)
                .setSource(mapping, XContentType.JSON)
                .get();
        return false;
    }
```

在数据库层面，封装 DAO 导入数据：

```
public class Dao {
    private Connection conn;
    public void getConnection(){
```

```
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            String user="root";
            String passwd="123456";
            String url="jdbc:mysql://localhost:3306/News";
            conn= DriverManager.getConnection(url,user,passwd);
            if (conn!=null){
                System.out.println("mysql 连接成功!");
            }else{
                System.out.println("mysql 连接失败!");
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void mysqlToEs(){
        String sql="SELECT * FROM news";
        TransportClient client= EsUtils.getSingleClient();
        try {
            PreparedStatement pstm=conn.prepareStatement(sql);
            ResultSet resultSet=pstm.executeQuery();
            Map<String,Object> map=new HashMap<String, Object>();
            while (resultSet.next()){
                int nid=resultSet.getInt(1);
                map.put("id",nid);
                map.put("title",resultSet.getString(2));
                map.put("key_word",resultSet.getString(3));
                map.put("content",resultSet.getString(4));
                map.put("url",resultSet.getString(5));
                map.put("reply",resultSet.getInt(6));
                map.put("source",resultSet.getString(7));
                String postdatetime=resultSet.getTimestamp(8)
                              .toString();
                map.put("postdate",postdatetime.substring(0,
                              postdatetime.length()-2));
                System.out.println(map);
                client.prepareIndex("spnews","news",String.valueOf(nid))
                        .setSource(map)
                        .execute()
                        .actionGet();
            }
```

```
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

执行数据导入的主函数:

```
    public static void main(String[] args) {
        //1.创建索引
        EsUtils.createIndex("spnews", 3, 0);
        //2.设置Mapping
        try {
            XContentBuilder builder = jsonBuilder()
                    .startObject().startObject("properties")
                    .startObject("id")
                    .field("type", "long")
                    .endObject()
                    .startObject("title")
                    .field("type", "text")
                    .field("analyzer", "ik_max_word")
                    .field("search_analyzer", "ik_max_word")
                    .endObject()
                    .startObject("key_word")
                    .field("type", "text")
                    .field("analyzer", "ik_max_word")
                    .field("search_analyzer", "ik_max_word")
                    .endObject()
                    .startObject("content")
                    .field("type", "text")
                    .field("analyzer", "ik_max_word")
                    .field("search_analyzer", "ik_max_word")
                    .endObject()
                    .startObject("url")
                    .field("type", "keyword")
                    .endObject()
                    .startObject("reply")
                    .field("type", "long")
                    .endObject()
                    .startObject("source")
                    .field("type", "keyword")
                    .endObject()
                    .startObject("postdate")
                    .field("type", "date")
```

```
                    .field("format", "yyyy-MM-dd HH:mm:ss")
                    .endObject()
                    .endObject()
                    .endObject();
            EsUtils.setMapping("spnews", "news", builder.string());
        } catch (IOException e) {
            e.printStackTrace();
        }
        //3. 读取 MySQL
        Dao dao = new Dao();
        dao.getConnection();
        dao.mysqlToEs();
    }
```

10.4　查询界面

这一节来实现一个用户查询的搜索框，首先创建一个 jsp 页面作为首页，一般命名为 index.jsp。在 index.jsp 中加入一个 form 表单，加入一个输入框和一个提交按钮。用户输入搜索关键词并单击搜索按钮时，要把用户输入的内容传给后台的 servlet，因此设置 form 表单的 action 的取值为后台的 servlet 名称。index.jsp 中的核心代码如代码清单 10-9 所示。

代码清单 10-9

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>新闻搜索</title>
    <link type="text/css" rel="stylesheet" href="css/index.css">
</head>
<body>
<div class="box">
    <h1>Elasticsearch 新闻搜索</h1>
    <div class="searchbox">
        <form action="/SearchNews" method="get">
            <input type="text" name="query">
            <input type="submit" value="搜索一下">
        </form>
    </div>
</div>
</body>
</html>
```

为了界面美观，增加样式表文件 index.css，内容如下：

```
body{
    margin: 0 auto;
}

.box{
    border: 1px solid #cccccc;
    width: 600px;
    height: 400px;
    margin: 100px auto;
}
.box h1{
    text-align: center;
    color: #008040;
    margin: 50px auto;
}

.searchbox{
    height: 30px;
    border: 1px solid #008040;
    width: 80%;
    margin:50px auto ;
}
.searchbox input[type="text"]{
    height: 30px;
    width: 85%;
    outline: none;
    border: 0;
    font-size: 18px;
}

.searchbox input[type="submit"]{
    height: 30px;
    width:14% ;
    float: right;
    border: 0;
    outline: none;
    background-color: #008040;
    color: #ffffff;
}
```

在 Tomcat 中运行，新闻搜索框效果如图 10-3 所示。

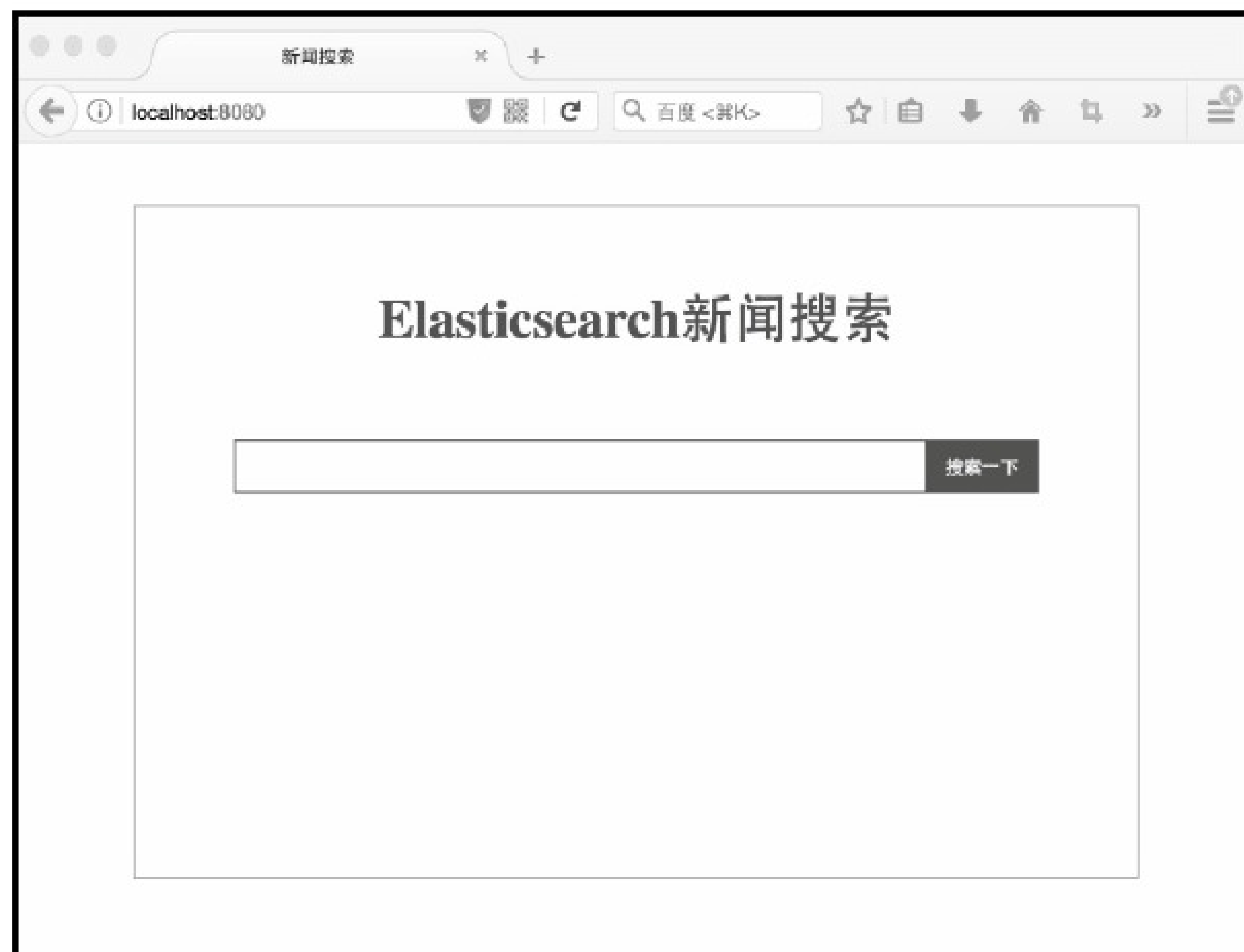


图 10-3　新闻搜索查询界面

10.5　搜索新闻

用户输入的查询语句需要提交给后台的 servlet 接收，servlet 采用注解的方式配置，使用 multiMatchQuery 对 title 和 content 字段进行搜索，同时对 title 和 content 字段进行高亮处理。

大型搜索引擎一次会检索到上百万条记录，根据用户习惯，一般只有前几页的内容会被用户点击查看，不可避免的需要使用到分页的功能。分页就是对搜索结果做分批处理，用户如果在其中没有找到想要的内容，可以通过制定页码或翻页的方式转换可见内容，直到找到需要的内容为止。这里设置每页返回 5 条新闻数据，最后把搜索到的数据放到 request 作用域中并转发至搜索结果页。

```
@WebServlet(name = "/SearchNews", urlPatterns = "/SearchNews")
public class SearchServlet extends HttpServlet {
   @Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
       req.setCharacterEncoding("UTF-8");
       String query = req.getParameter("query");
       System.out.println(query);
       String pageNumStr=req.getParameter("pageNum");
       int pageNum=1;
       if (pageNumStr!=null&&Integer.parseInt(pageNumStr)>1){
          pageNum=Integer.parseInt(pageNumStr);
       }
       searchSpnews(query, pageNum,req);
       req.setAttribute("queryBack", query);
       req.getRequestDispatcher("result.jsp").forward(req, resp);
```

```
}
private void searchSpnews(String query, int pageNum,HttpServletRequest
     req) {
    long start = System.currentTimeMillis();
    TransportClient client = EsUtils.getSingleClient();
    MultiMatchQueryBuilder multiMatchQuery = QueryBuilders
            .multiMatchQuery(query, "title", "content");
    HighlightBuilder highlightBuilder = new HighlightBuilder()
            .preTags("<span style=\"color:red\">")
            .postTags("</span>")
            .field("title")
            .field("content");

    SearchResponse searchResponse = client.prepareSearch("spnews")
            .setTypes("news")
            .setQuery(multiMatchQuery)
            .highlighter(highlightBuilder)
            .setFrom((pageNum-1)*5)
            .setSize(5)
            .execute()
            .actionGet();
    SearchHits hits = searchResponse.getHits();
    ArrayList<Map<String, Object>> newslist = new ArrayList<Map<String,
         Object>>();
    for (SearchHit hit : hits) {
        Map<String, Object> news = hit.getSourceAsMap();
        HighlightField hTitle = hit.getHighlightFields().get("title");
        if (hTitle != null) {
            Text[] fragments = hTitle.fragments();
            String hTitleStr = "";
            for (Text text : fragments) {
                hTitleStr += text;
            }
            news.put("title", hTitleStr);
        }

        HighlightField hContent = hit.getHighlightFields().get("content");
        if (hContent != null) {
            Text[] fragments = hContent.fragments();
            String hContentStr = "";
            for (Text text : fragments) {
                hContentStr += text;
            }
            news.put("content", hContentStr);
```

```
            }
            newslist.add(news);
        }
        long end = System.currentTimeMillis();
        req.setAttribute("newslist", newslist);
        req.setAttribute("totalHits", hits.getTotalHits() + "");
        req.setAttribute("totalTime", (end - start) + "");
    }
    @Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
        doGet(req, resp);
    }
}
```

10.6　结果展示

编辑 result.jsp，将查询结果展示到 jsp 页面中。通过 request 对象的 getAttribute()方法获取查询结果，遍历集合，把新闻内容输出到页面中。代码如下：

```
<%@ page import="java.util.ArrayList" %>
<%@ page import="java.util.Map" %>
<%@ page import="java.util.Iterator" %>
<%@ page import="com.sun.org.apache.bcel.internal.generic.IF_ACMPEQ" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%
String queryBack = (String) request.getAttribute("queryBack");
ArrayList<Map<String, Object>> newslist = (ArrayList<Map<String, Object>>)
    request.getAttribute("newslist");
    String totalHits = (String) request.getAttribute("totalHits");
    String totalTime = (String) request.getAttribute("totalTime");
    int pages = Integer.parseInt(totalHits) / 10 + 1;
    pages = pages > 10 ? 10 : pages;
%>
<html>
<head>
    <title>搜索结果</title>
    <link type="text/css" rel="stylesheet" href="css/result.css">
</head>
<body>
<div class="result_search">
    <div class="logo"> <h2><a href="index.jsp">新闻搜索</a></h2></div>
```

```
        <div class="searchbox">
            <form action="/SearchNews" method="get">
                <input type="text" name="query" value="<%=queryBack%>">
                <input type="submit" value="搜索一下">
            </form>
        </div>
    </div>
    <h5 class="result_info">共搜索到<span><%=totalHits%></span>条结果,耗时<span>
    <%=Double.parseDouble(totalTime) / 1000.0 %></span>秒
    </h5>
    <div class="newslist">
        <%
            if (newslist.size() > 0) {
                Iterator<Map<String, Object>> iter = newslist.iterator();
                while (iter.hasNext()) {
                    Map<String, Object> news = iter.next();
                    String content = news.get("content").toString();
                    content = content.length() > 200 ? content.substring(0, 200) :
                            content;
        %>
        <div class="news">
            <h4><a href="<%=news.get("url")%>"><%=news.get("title")%> </a></h4>
            <p><%=content%> </p>
        </div>
        <%
                }
            }
        %>
    </div>
    <div class="page">
        <ul>
          <% for (int i = 1; i <= pages; i++) { %>
          <li><a href="/SearchNews?query=<%=queryBack%>&pageNum=<%=i%>"><%=i%>
          </a></li>
          <% } %>
        </ul>
    </div>
    <div class="info">
        <p>新闻搜索项目实战 Powered By <b> Elasticsearch</b></p>
        <p>@2017 All right reserved</p>
    </div>
    </body>
    </html>
```

搜索结果页的样式表文件 result.css 的内容如下：

```
body {
    margin: 0;
    padding: 0;
}

.result_search {
    width: 100%;
    height: 60px;
    background-color: #cccccc;
}

.logo {
    float: left;
}

.logo h2 {
    color: #008040;
    padding-left: 20px;
}
.logo h2 a:link,
.logo h2 a:visited{
     text-decoration: none;
     color: #008040;
}
.searchbox {
   float: left;
   border: 1px solid #008040;
   height: 30px;
   width: 500px;
   margin-top: 15px;
   margin-left: 30px;
}
.searchbox input[type="text"] {
   width: 85%;
   height: 30px;
   border: 0;
   outline: none;
   font-size: 18px;
}

.searchbox input[type="submit"] {
   width: 15%;
   border: 0;
   outline: none;
```

```
    height: 30px;
    background-color: #008040;
    color: #ffffff;
    float: right;
}

.result_info {
    margin-left: 30px;
}

.result_info span {
    color: #ff0000;
}

.newslist {
    width: 700px;
}

.newslist h4 {
    padding: 0;
    margin: 0;
    margin-left: 20px;
}

.newslist h4 a:link, .newslist h4 a:visited {
     text-decoration: none;
}
.newslist h4 a:hover {
    text-decoration: underline;
}

.newslist p {
    margin-left: 30px;
    line-height: 1.5;
    font-size: 13px;
}

.page {
    margin-left: 50px;
    height: 30px;
}

.page ul li {
    list-style: none;
    float: left;
    width: 50px;
}
```

```
.page ul li a:link, .page ul li a:visited {
    text-decoration: none;
}

.info {
    width: 800px;
}

.info p {
    text-align: center;
   font-size: 12px;
   color: #808080;
}
```

最后，在 Tomcat 中运行项目，输入关键词“奥运会”进行测试，搜索结果如图 10-4 所示。

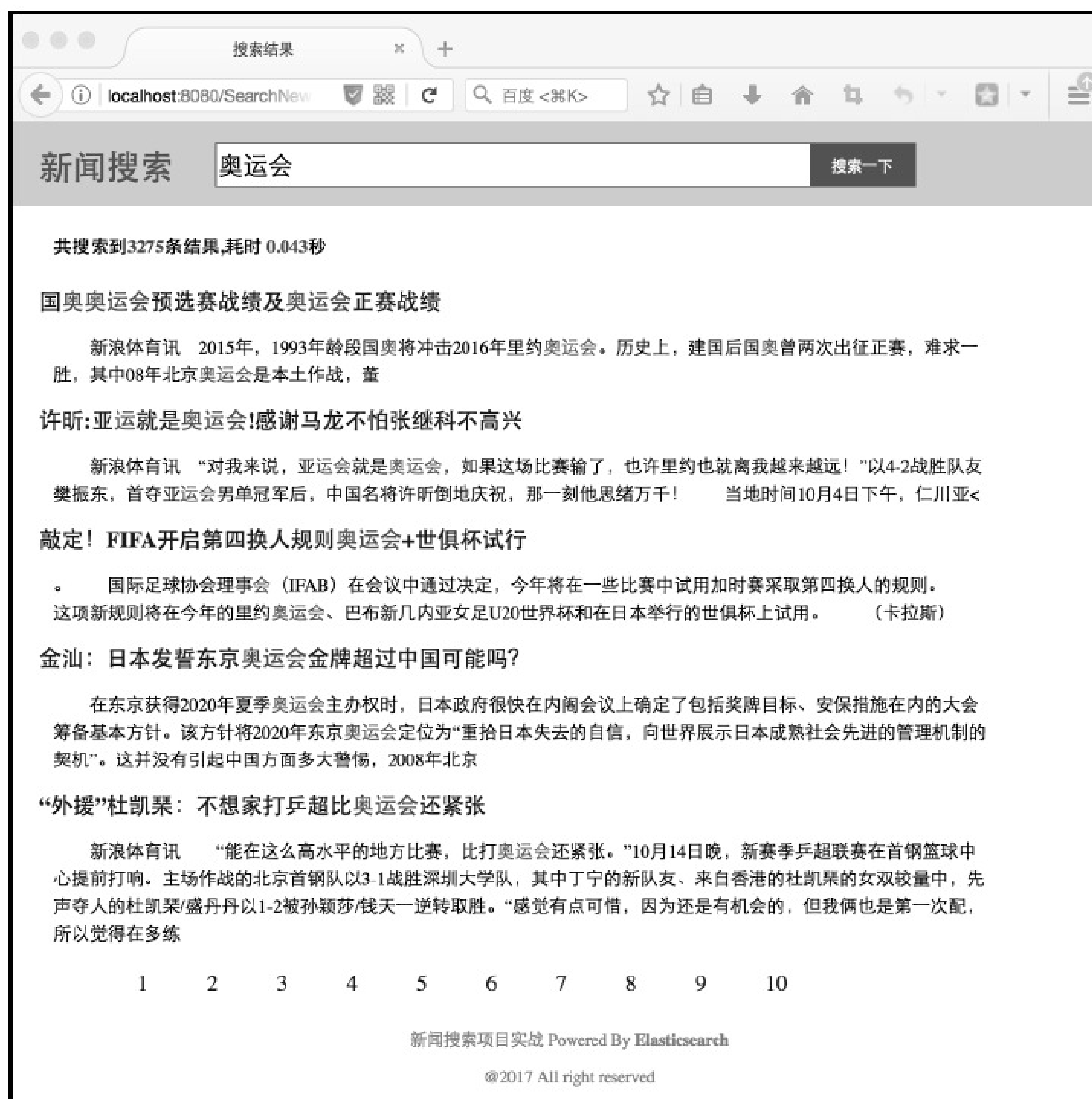


图 10-4　新闻搜索结果页面

10.7　本章小结

本章通过实现对 MySQL 中的表进行全文检索这一需求，贯穿了 MySQL、JDBC、Elasticsearch Java API 以及 Java Web 的相关知识，通过实际项目加深读者对 Elasticsearch 的理解和运用。

第 11 章

Elasticsearch For Hadoop

本章学习要点：

- ★ Hadoop 基本配置
- ★ ES-Hadoop 安装
- ★ 从 HDFS 到 Elasticsearch
- ★ 从 Elasticsearch 到 HDFS

众所周知，Hadoop 在离线批处理程序上有着天然的优势，但是也存在实时性差的致命缺陷。Elasticsearch for Apache Hadoop 是一个用于 Elasticsearch 和 Hadoop 进行交互的开源独立库，简称 ES-Hadoop，在 Hadoop 和 Elasticsearch 之间起到桥梁的作用，完美地把 Hadoop 的批处理优势和 Elasticsearch 强大的全文检索引擎结合起来。ES-Hadoop 开辟了更加广阔的应用空间，通过 ES-Hadoop 可以索引 Hadoop 中的数据到 Elasticsearch，充分利用其查询和聚合分析功能，也可以在 Kibana 中做进一步的可视化分析，同时也可以把 Elasticsearch 中的数据放到 Hadoop 生态系统中做运算，ES-Hadoop 支持 Spark、Streaming、SparkSQL，除此之外，不论你是使用 Hive、Pig、Storm、Cascading 还是运行单独的 Map/Reduce，ES-Hadoop 提供的接口都支持从 Elasticsearch 中进行索引和查询操作。图 11-1 很好地说明了 ES-Hadoop 和大数据生态系统之间的关系。

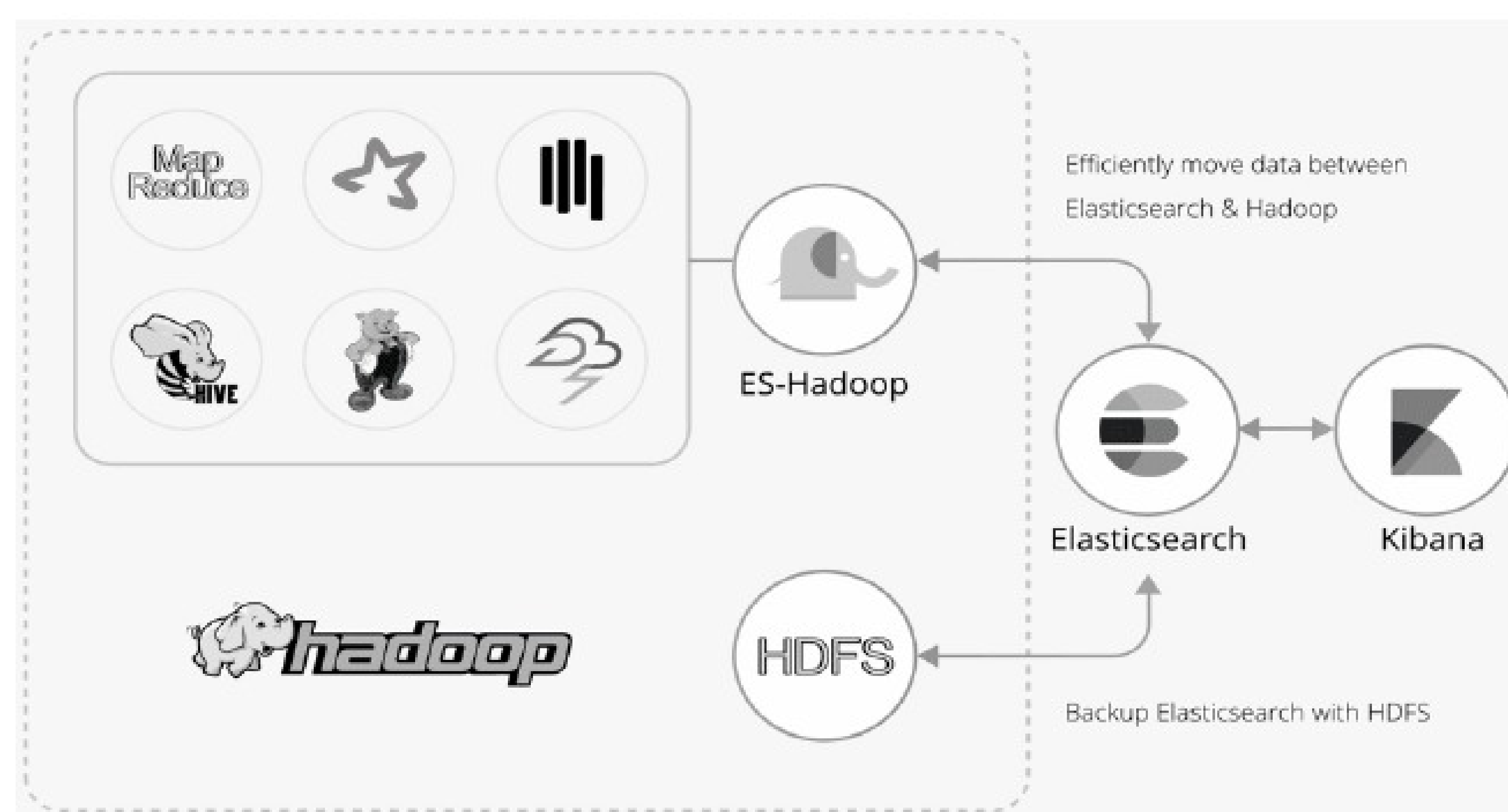


图 11-1　ES-Hadoop 与大数据的关系

11.1　Hadoop 基础

为了方便不熟悉 Hadoop 的读者，本小节先介绍一些 Hadoop 的基础知识，包括如何在 Linux 机器上安装配置 Hadoop 伪分布式集群、HDFS 常用命令。

11.1.1　SSH 配置

Hadoop 使用 SSH 进行通信，设置密码为空，免去每次通信都需要密码，SSH 免密码登录配置的步骤如下：

步骤 01 打开 terminal，进入根目录，运行命令：

```
cd
```

步骤 02 如果机器之前没有 SSH 相关配置，没有.ssh 文件夹，显示隐藏文件命令：

```
ls -a
```

步骤 03 生成密钥，执行命令如下：

```
ssh-keygen -t rsa -P ""
```

执行过程如图 11-2 所示。

```
2. bash
Bee:~ bee$ cd
Bee:~ bee$ ssh-keygen -t rsa -P ""
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/bee/.ssh/id_rsa):
Created directory '/Users/bee/.ssh'.
Your identification has been saved in /Users/bee/.ssh/id_rsa.
Your public key has been saved in /Users/bee/.ssh/id_rsa.pub.
The key fingerprint is:
99:99:2e:fa:29:aa:94:bd:94:5f:65:db:50:4d:74:41 bee@Bee.local
The key's randomart image is:
+--[ RSA 2048]----+
|            .o.E.|
|            o .  |
|           . .   |
|         =.      |
|        S+       |
|  o .  .o +      |
| o +  .... .     |
|. . +..o         |
|...o.+o          |
+-----------------+
Bee:~ bee$
```

11-2　生成密钥

步骤 04 执行完成以后会生成.ssh 文件夹，在.ssh 文件夹中会生成一对密钥，id_rsa 是生成的私钥文件，id_rsa.pub 是生成的公钥文件。把公钥文件复制到本机的 authorized_keys 文件中，执行命令：

```
cat id_rsa.pub >> authorized_keys
```

步骤 05 测试 ssh 登录，执行命令：

```
ssh localhost
```

第一次登录会把本机 ip 加入一个 known_hosts 文件中，known_hosts 文保存了对所有用户都可信赖的远程主机的公钥，如图 11-3 所示，再次登录就无须密码验证了。

```
2. ssh
Bee:~ bee$ cd .ssh/
Bee:.ssh bee$ ls
id_rsa          id_rsa.pub
Bee:.ssh bee$ cat id_rsa.pub >> authorized_keys
Bee:.ssh bee$ ls
authorized_keys id_rsa          id_rsa.pub
Bee:.ssh bee$ ssh localhost
The authenticity of host 'localhost (127.0.0.1)' can't be established.
RSA key fingerprint is 5e:4e:fe:e7:44:48:f6:d4:bb:a1:01:af:37:fc:e4:fe.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
Last login: Mon Aug 21 17:43:50 2017 from localhost
Bee:~ bee$ ssh localhost
Last login: Mon Aug 21 17:45:06 2017 from localhost
Bee:~ bee$
```

图 11-3　免密码登录

11.1.2　Hadoop 下载

Hadoop 下载地址：http://apache.org/dist/hadoop/common，选择 2.7.3 版本，选择下载 hadoop-2.7.3.tar.gz，下载后解压缩到指定目录，解压命令如下：

```
sudo tar -zxvf hadoop-2.7.3.tar.gz
```

11.1.3　Hadoop 单机模式

Hadoop 有 3 种安装模式：单机模式、伪分布式模式、完全分布式模式。解压 Haoop 安装文件之后即可运行单机模式，运行 wordcount 测试是否安装成功，步骤如下：

步骤 01　在 hadoop-2.7.3 目录下新建 input 文件夹：

```
sudo mkdir input
```

步骤 02　在 input 文件夹下新增 2 个文本文件并写入单词用于测试：

```
echo 'hello world' > file1.txt
echo 'hello hadoop' > file2.txt
```

步骤 03　运行 wordcount 例子：

```
sudo ./bin/hadoop jar ./share/hadoop/mapreduce/hadoop-
mapreduce-examples-2.7.3.jar wordcount input/ output
```

步骤 04　查看运行结果：

```
cat output/part-r-00000
```

统计结果：

```
hadoop  1
hello   2
```

```
world   1
```

11.1.4　Hadoop 伪分布式模式

Hadoop 伪分布式模式主要涉及以下配置信息：

- 修改 HADOOP_OPTS。
- 修改 core-site.xml 配置 HDFS 地址和端口。
- 修改 mapred-site.xml 文件配置 JobTracker 的地址和端口。
- 修改 hdfs-site.xml 文件配置 HDFS 的副本数。

hadoop-env.sh、core-site.xml、mapred-site.xml、hdfs-site.xml 都位于 hadoop-2.7.3/etc/hadoop 目录下。

步骤 01 编辑 hadoop-env.sh，找到 HADOOP_OPTS 并将其注释：

```
#export HADOOP_OPTS="$HADOOP_OPTS -Djava.net.preferIPv4Stack=true"
```

修改为：

```
export HADOOP_OPTS="$HADOOP_OPTS -Djava.net.preferIPv4Stack=true
    -Djava.security.krb5.realm= -Djava.security.krb5.kdc="
```

步骤 02 编辑 core-site.xml，修改为如下配置：

```
<configuration>
  <property>
      <name>hadoop.tmp.dir</name>
      <value>/usr/local/Cellar/hadoop-2.7.3/hdfs/tmp</value>
      <description>A base for other temporary directories</description>
  </property>
  <property>
      <name>fs.default.name</name>
      <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

其中/usr/local/Cellar/hadoop-2.7.3/hdfs/tmp 可以自定义，fs.default.name 保存了 NameNode 的位置，HDFS 和 MapReduce 组件都需要用到它。

步骤 03 编辑 mapred-site.xml.template，增加配置信息如下：

```
<configuration>
    <property>
      <name>mapred.job.tracker</name>
      <value>localhost:9010</value>
  </property>
</configuration>
```

变量 mapred.job.tracker 保存了 JobTracker 的位置，因为只有 MapReduce 组件需要知道这个位置，所以它出现在 mapred-site.xml 文件中。

步骤 04 编辑 hdfs-site.xml：

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

变量 dfs.replication 指定了每个 HDFS 数据库的复制次数。通常为 3，由于我们只有一台主机和一个伪分布式的 DataNode，因此将此值修改为 1。

通过上面的配置，Hadoop 伪分布式配置就已经完成了，接下来启动 Hadoop，首先格式化 namenode，执行命令：

```
./bin/hadoop namenode -format
```

格式化过程如图 11-4 所示。

```
1. bash
Bee:hadoop-2.7.3 bee$ ./bin/hadoop namenode -format
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

17/08/21 22:36:26 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = 115.1.168.192.in-addr.arpa/203.208.39.104
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 2.7.3
STARTUP_MSG:   classpath = /usr/local/Cellar/hadoop-2.7.3/etc/hadoop:/usr/local/
Cellar/hadoop-2.7.3/share/hadoop/common/lib/activation-1.1.jar:/usr/local/Cellar
/hadoop-2.7.3/share/hadoop/common/lib/apacheds-i18n-2.0.0-M15.jar:/usr/local/Cel
lar/hadoop-2.7.3/share/hadoop/common/lib/apacheds-kerberos-codec-2.0.0-M15.jar:/
usr/local/Cellar/hadoop-2.7.3/share/hadoop/common/lib/api-asn1-api-1.0.0-M20.jar
:/usr/local/Cellar/hadoop-2.7.3/share/hadoop/common/lib/api-util-1.0.0-M20.jar:/
usr/local/Cellar/hadoop-2.7.3/share/hadoop/common/lib/asm-3.2.jar:/usr/local/Cel
lar/hadoop-2.7.3/share/hadoop/common/lib/avro-1.7.4.jar:/usr/local/Cellar/hadoop
-2.7.3/share/hadoop/common/lib/commons-beanutils-1.7.0.jar:/usr/local/Cellar/had
oop-2.7.3/share/hadoop/common/lib/commons-beanutils-core-1.8.0.jar:/usr/local/Ce
```

图 11-4 格式化 namenode

执行完成以后，启动 Hadoop，命令如下：

```
./sbin/start-all.sh
```

执行 start-all.sh 脚本和先执行 start-dfs.sh 再执行 start-yarn.sh 是一样的，运行结果如图 11-5 所示。

启动成功之后，使用 jps 命令查看 JVM 进程，正常情况下如果看到 NameNode、DataNode、ResourceManager、NodeManager 和 SecondaryNameNode，就说明配置成功。

```
1. bash
Bee:hadoop-2.7.3 bee$ ./sbin/start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
17/08/21 23:56:14 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform.
.. using builtin-java classes where applicable
Starting namenodes on [localhost]
localhost: starting namenode, logging to /usr/local/Cellar/hadoop-2.7.3/logs/hadoop-bee-namenode-Bee.
local.out
localhost: starting datanode, logging to /usr/local/Cellar/hadoop-2.7.3/logs/hadoop-bee-datanode-Bee.
local.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/Cellar/hadoop-2.7.3/logs/hadoop-bee-second
arynamenode-Bee.local.out
17/08/21 23:56:30 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform.
.. using builtin-java classes where applicable
starting yarn daemons
starting resourcemanager, logging to /usr/local/Cellar/hadoop-2.7.3/logs/yarn-bee-resourcemanager-Bee
.local.out
localhost: starting nodemanager, logging to /usr/local/Cellar/hadoop-2.7.3/logs/yarn-bee-nodemanager-
Bee.local.out
Bee:hadoop-2.7.3 bee$ jps
5808 SecondaryNameNode
6052 NodeManager
5576 NameNode
6088 Jps
5946 ResourceManager
5676 DataNode
```

图 11-5　启动 Hadoop

最后，访问 HDFS 的 50070 端口，可以看到如图 11-6 所示的界面。

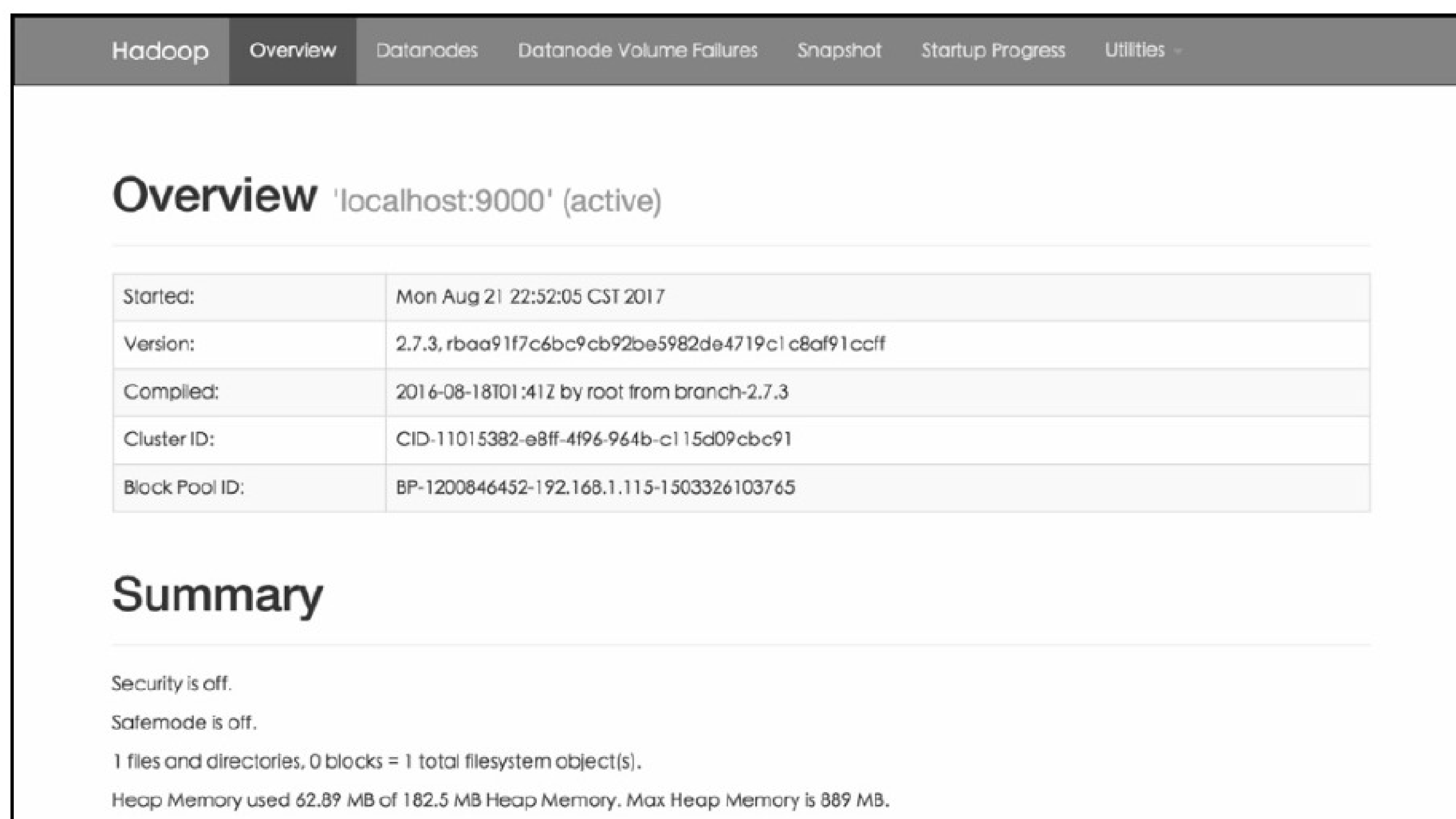


图 11-6　访问 HDFS 的 WEB 端口

为了使用方便，把 Hadoop 的执行脚本加入系统环境变量中，后期就不需要每次都切换到 Hadoop 安装目录执行 Hadoop 相关命令了。编辑/etc/profile，加入如下配置：

```
export HADOOP_HOME=/usr/local/Cellar/hadoop-2.7.3
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

其中 HADOOP_HOME 是 Hadoop 安装包所在的位置，执行 source 命令使配置生效：

```
source /etc/profile
```

11.1.5　HDFS 常用操作

在浏览器查看 HDFS 文件，如图 11-7 所示。

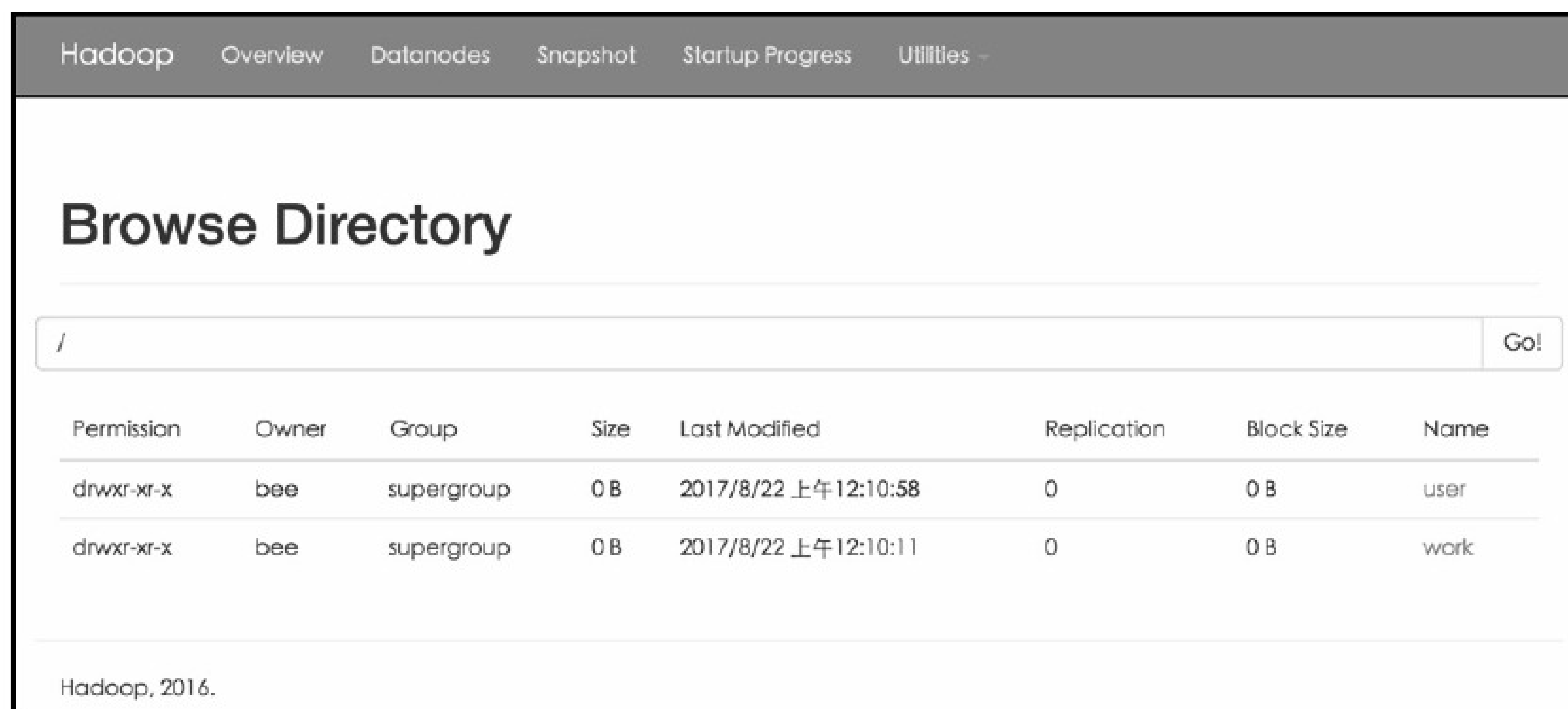


图 11-7　在浏览器查看 HDFS 文件

查看 HDFS 根目录下的文件：

```
hadoop fs -ls /
```

在 HDFS 之上新建一个文件夹：

```
hadoop fs  -mkdir /work
```

递归创建多级文件夹：

```
hadoop fs -mkdir -p /a/b/c
```

上传本地文件到 HDFS：

```
hadoop fs -put a.txt /work
```

检查文件是否存在：

```
hadoop fs -test -e /work/a.txt
echo $?(打印结果为 0 说明文件存在，为 1 说明文件不存在)
```

查看 HDFS 中的文件内容：

```
hadoop fs -cat /work/a.txt
```

删除 HDFS 上的文件：

```
hadoop fs -rm /work/a.txt
```

删除 HDFS 上的文件夹：

```
hadoop fs -rmr /work/a
```

追加到文件末尾：

```
hadoop fs -appendToFile local.txt /work/a.txt(local.txt 中的内容追加到 a.txt 中)
hdfs dfsadmin -report
```

11.2　ES-Hadoop 安装

ES-Hadoop 有直接下载和使用 Maven 管理两种方式，Elasticsearch 5.0 之后 Elastic 公司对所有的软件做了统一版本号处理，因此 ES-Hadoop 的版本也要和 Elasticsearch 版本一致。ES-Hadoop 5.4 版本的运行同样需要 JDK 版本不能低于 1.8，ES-Hadoop 不支持 Elasticsearch 1.0 之前的版本。

11.2.1　压缩包下载

ES-Hadoop 下载地址：https://www.elastic.co/downloads/hadoop，下载并解压后，把 dist 目录下的 jar 包导入 hadoop 即可。

为了方便程序的运行，把 ES-HADOOP 的 jar 包导入环境变量，编辑/etc/profile，添加以下内容：

```
export EsHadoop_HOME=/usr/local/Cellar/elasticsearch-hadoop-5.4.0
export CLASSPATH=$CLASSPATH:$EsHadoop_HOME/dist/*
```

最后使 profile 文件生效：

```
source /etc/profile
```

11.2.2　Maven 依赖

如果使用 Maven 管理 jar 包，就可以在 pom.xml 文件中加入以下依赖：

```
<dependency>
  <groupId>org.elasticsearch</groupId>
  <artifactId>elasticsearch-hadoop</artifactId>
  <version>5.4.0</version>
</dependency>
```

上面的依赖包含了 MapReduce、Pig、Hive、Spark 等完整的依赖，如果只想单独使用某一个功能，可以细化后分别加入。

支持 Map/Reduce 的最小依赖：

```
<dependency>
  <groupId>org.elasticsearch</groupId>
  <artifactId>elasticsearch-hadoop-mr</artifactId>
  <version>5.4.0</version>
</dependency>
```

支持 Apache Hive 的最小依赖：

```
<dependency>
  <groupId>org.elasticsearch</groupId>
```

```
  <artifactId>elasticsearch-hadoop-hive</artifactId>
  <version>5.4.0</version>
</dependency>
```

支持 Apache Pig 的最小依赖：

```
<dependency>
  <groupId>org.elasticsearch</groupId>
  <artifactId>elasticsearch-hadoop-pig</artifactId>
  <version>5.4.0</version>
</dependency>
```

支持 Apache Spark 的最小依赖：

```
<dependency>
  <groupId>org.elasticsearch</groupId>
  <artifactId>elasticsearch-spark-20_2.10</artifactId>
  <version>5.4.0</version>
</dependency>
```

支持 Storm 的最小依赖：

```
<dependency>
  <groupId>org.elasticsearch</groupId>
  <artifactId>elasticsearch-storm</artifactId>
  <version>5.4.0</version>
</dependency>
```

11.3　从 HDFS 到 Elasticsearch

这一节介绍使用 MapReduce 把 HDFS 上的数据批量导入 Elasticsearch 的方法和步骤。由于 MapReduce 是按行读取数据的，因此 HDFS 上的数据每一行都要是 json 格式的字符串。

11.3.1　测试数据

准备一些测试数据，数据内容为 json 格式，每行是一条文档。把下列内容保存到 blog.json 中。

```
{"id":"1","title":"git 简介","posttime":"2016-06-11","content":"svn 与 git 的
最主要区别..."}
{"id":"2","title":"ava 中泛型的介绍与简单使用","posttime":"2016-06-12",
"content":"基本操作：CRUD ..."}
{"id":"3","title":"SQL 基本操作","posttime":"2016-06-13","content":"svn 与
git 的最主要区别..."}
{"id":"4","title":"Hibernate 框架基础","posttime":"2016-06-14","content":
"Hibernate 框架基础..."}
```

```
{"id":"5","title":"Shell 基本知识","posttime":"2016-06-15","content":"Shell 是什么..."}
```

启动 Hadoop 和 HDFS，执行以下命令上传 blog.json 到 HDFS：

```
hadoop fs -put blog.json /work
```

11.3.2　编写程序

下面开始编写 ES-Hadoop 程序，推荐使用 maven 来进行项目管理，操作步骤如下：

步骤 01 安装 maven。首先确保计算机已经正确安装安装 maven。

步骤 02 生成工程框架。在工程根目录处执行如下命令：

```
mvn archetype:generate -DgroupId=com.tsinghua -DartifactId=eshadoop
-DarchetypeArtifactId = maven-archetype-quickstart -DinteractiveMode=false
```

命令执行过程如图 11-8 所示，maven 会自动生成一个空的 Sample 工程，工程名为 emrtoes（和指定的 artifactId 一致），里面包含一个简单的 pom.xml 和 App 类（类的包路径和指定的 groupId 一致）。

```
3. bash
Bee:elk bee$ mvn -v
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-11T00:41:47+08:00)
Maven home: /usr/local/Cellar/apache-maven-3.3.9
Java version: 1.8.0_121, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/jre
Default locale: zh_CN, platform encoding: UTF-8
OS name: "mac os x", version: "10.13.3", arch: "x86_64", family: "mac"
Bee:elk bee$ mvn archetype:generate -DgroupId=com.tsinghua -DartifactId=eshadoop -DarchetypeArtifactId=
maven-archetype-quickstart -DinteractiveMode=false
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:3.0.1:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:3.0.1:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:3.0.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-quicks
tart:1.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: basedir, Value: /Users/bee/Documents/elk
[INFO] Parameter: package, Value: com.tsinghua
[INFO] Parameter: groupId, Value: com.tsinghua
[INFO] Parameter: artifactId, Value: eshadoop
[INFO] Parameter: packageName, Value: com.tsinghua
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /Users/bee/Documents/elk/eshadoop
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 12.078 s
[INFO] Finished at: 2018-05-03T14:17:13+08:00
[INFO] Final Memory: 17M/209M
[INFO] ------------------------------------------------------------------------
```

图 11-8　命令创建 maven 工程

步骤 03 加入 Hadoop 和 ES-Hadoop 依赖。使用 IDEA 打开这个工程，编辑 pom.xml 文件。在 dependencies 内添加如下内容：

```
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-mapreduce-client-common</artifactId>
  <version>2.7.3</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-common</artifactId>
  <version>2.7.3</version>
</dependency>
<dependency>
  <groupId>org.elasticsearch</groupId>
  <artifactId>elasticsearch-hadoop-mr</artifactId>
  <version>5.4.0</version>
</dependency>
```

步骤 04 添加打包插件。由于使用了第三方库，需要把第三方库打包到 jar 文件中，在 pom.xml 中添加 maven-assembly-plugin 插件的坐标：

```
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <archive>
            <manifest><mainClass>com.tsinghua.HdfsToEs</mainClass>
</manifest>
          </archive>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
        <executions>
          <execution>
            <id>make-assembly</id>
            <phase>package</phase>
            <goals> <goal>single</goal></goals>
          </execution>
        </executions>
      </plugin>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
```

```
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.0.0</version>
            <executions>
              <execution>
                <phase>package</phase>
                <goals><goal>shade</goal> </goals>
                <configuration>
                  <transformers>
                    <transformer implementation="org.apache.maven.plugins.shade
                            .resource.ManifestResourceTransformer">
                      <mainClass>com.tsinghua.HdfsToEs</mainClass>
                    </transformer>
                    <transformer implementation="org.apache.maven.plugins.shade
                        .resource.ApacheLicenseResourceTransformer">
                    </transformer>
                  </transformers>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
```

注意，mainClass 中对应的类是 ES-Hadoop 作业的主程序。如果类路径不一致，每次打包前需要修改。

步骤 05 编写代码。在 com. tsinghua 包下和 App 类平行的位置添加新类 HdfsToEs.java。由于只需要按行读取 HDFS 上的文件内容然后写入 Elasticsearch，没有 Reduce 过程，程序中只需要有 Map 过程，完整的代码如下：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.elasticsearch.hadoop.mr.EsOutputFormat;
import java.io.IOException;
public class HdfsToEs {
   public static class MyMapper extends Mapper<Object,
        Text, NullWritable, Text> {
       private Text line = new Text();
       @Override
```

```
        protected void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            if (value.getLength() > 0) {
                line.set(value);
                context.write(NullWritable.get(), line);
            }
        }
    }

    public static void main(String[] args) throws IOException,
        ClassNotFoundException, InterruptedException {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
         args) .getRemainingArgs();
         conf.setBoolean("mapred.map.tasks.speculative.execution", false);
         conf.setBoolean("mapred.reduce.tasks.speculative.execution", false);
         conf.set("es.nodes", "127.0.0.1");
         conf.set("es.port", "9200");
         conf.set("es.nodes.wan.only", "true");
         conf.set("es.resource", "blog/csdn");
         conf.set("es.mapping.id", "id");
         conf.set("es.input.json", "yes");
         Job job = Job.getInstance(conf, "EmrToES");
         job.setJarByClass(HdfsToEs.class);
         job.setMapperClass(MyMapper.class);
         job.setInputFormatClass(TextInputFormat.class);
         job.setOutputFormatClass(EsOutputFormat.class);
         job.setMapOutputKeyClass(NullWritable.class);
         job.setMapOutputValueClass(Text.class);

         FileInputFormat.setInputPaths(job, new Path(otherArgs[0]));
         System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

步骤 06 编译并打包。在工程的目录下，执行如下命令：

```
mvn clean package
```

执行完毕以后，可在工程目录的 target 目录下看到一个 eshadoop -1.0-SNAPSHOT-jar- with -dependencies.jar，这个就是作业 jar 包。

步骤 07 执行 ES-Hadoop 作业。运行命令：

```
hadoop jar target/eshadoop-1.0-SNAPSHOT-jar-with-dependencies.jar
/work/blog.json
```

如果一切顺利，你会看到以下输出：

```
18/05/03 16:18:52 INFO Configuration.deprecation: session.id is deprecated.
Instead, use dfs.metrics.session-id
......
Elasticsearch Hadoop Counters
    Bulk Retries=0
    Bulk Retries Total Time(ms)=0
      .....
    Network Retries=0
    Network Total Time(ms)=132
    Node Retries=0
    Scroll Total=0
    Scroll Total Time(ms)=0
```

步骤 08 在 Elasticsearch 中查看执行结果。如图 11-9 所示，Elasticsearch 中查询 blog 索引返回数据说明执行成功。



图 11-9　ES-Hadoop 工程截图

为了方便理解，在 IDEA 中完整的工程目录截图如图 11-10 所示。

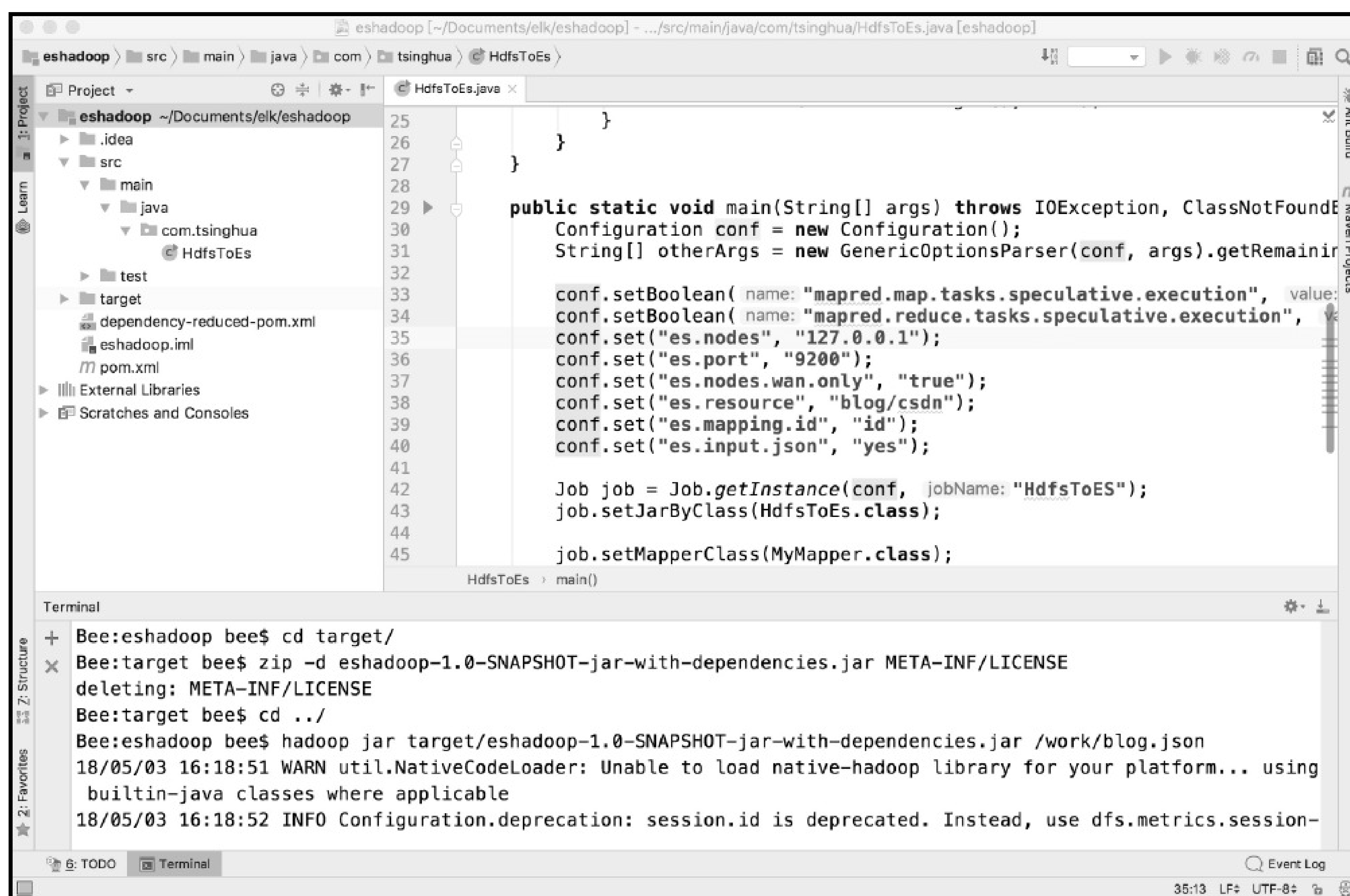


图 11-10　ES-Hadoop 工程截图

11.3.3　代码分析

按行读入 Map 过程，input kye 的类型为 Object，input value 的类型为 Text。输出的 key 为 NullWritable 类型，NullWritable 是 Writable 的一个特殊类，实现方法为空实现，不从数据流中读数据，也不写入数据，只充当占位符。MapReduce 中如果不需要使用键或值，就可以将键或值声明为 NullWritable，这里把输出的 key 设置为 NullWritable 类型。输出为 BytesWritable 类型，把 json 字符串序列化。

在 main 函数中，首先创建了 Configuration()类的一个对象 conf，通过 conf 配置一些参数。

- conf.setBoolean("mapred.map.tasks.speculative.execution", false);
  关闭 mapper 阶段的执行推测。
- conf.setBoolean("mapred.reduce.tasks.speculative.execution", false);
  关闭 reducer 阶段的执行推测。
- conf.set("es.nodes", "127.0.0.1");
  配置 Elasticsearch 的 IP。
- conf.set("es.port", "9200");
  配置 Elasticsearch 的端口。
- conf.set("es.resource", "blog/csdn");
  设置索引到 Elasticsearch 的索引名和类型名。
- conf.set("es.mapping.id", "id");
  设置文档 id，这个参数 id 是文档中的 id 字段。

- conf.set("es.input.json", "yes");
  指定输入的文件类型为 json。
- job.setInputFormatClass(TextInputFormat.class);
  设置输入流为文本类型。
- job.setOutputFormatClass(EsOutputFormat.class);
  设置输出为 EsOutputFormat 类型。
- job.setMapOutputKeyClass(NullWritable.class);
  设置 Map 的输出 key 类型为 NullWritable。
- job.setMapOutputValueClass(BytesWritable.class);
  设置 Map 的输出 value 类型为 BytesWritable。

11.4　从 Elasticsearch 到 HDFS

这一节介绍查询 Elasticsearch 中的数据并写入 HDFS 的方法和步骤。

11.4.1　读取索引到 HDFS

首先介绍读取 Elasticsearch 一个类型中的全部数据到 HDFS，这里读取索引为 blog 类型为 csdn 的所有文档，完整的代码如下：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.elasticsearch.hadoop.mr.EsInputFormat;
import java.io.IOException;
public class EsToHDFS {
   public static class MyMapper extends
Mapper<Writable,Writable,NullWritable,Text>{
       @Override
       protected void map(Writable key, Writable value, Context
          context) throws IOException, InterruptedException {
         Text text=new Text();
         text.set(value.toString());
         context.write(NullWritable.get(),text);
       }
```

```
    }
    public static void main(String[] args) throws Exception {
        Configuration configuration=new Configuration();
        String[] otherArgs = new GenericOptionsParser(configuration,
                                args).getRemainingArgs();
        configuration.set("es.nodes", "127.0.0.1");
        configuration.set("es.port", "9200");
        configuration.set("es.resource", "blog/csdn");
        configuration.set("es.output.json", "true");
        Job job = Job.getInstance(configuration,
            "hadoop es write test");
        job.setMapperClass(MyMapper.class);
        job.setNumReduceTasks(1);
        job.setMapOutputKeyClass(NullWritable.class);
        job.setMapOutputValueClass(Text.class);
        job.setInputFormatClass(EsInputFormat.class);
        FileOutputFormat.setOutputPath( job,new Path(otherArgs[0]));
        job.waitForCompletion(true);
    }
}
```

打包后执行作业：

```
mv target/eshadoop-1.0-SNAPSHOT-jar-with-dependencies.jar target/
EsToHDFS.jar
hadoop jar target/EsToHDFS.jar /work/blog_csdn
```

作业执行完成以后，会在 HDFS 的/work 目录下新增一个 blog_csdn 目录，使用 cat 命令查看数据，命令如下，执行结果如图 11-11 所示。

```
hadoop fs -cat /work/blog_csdn/part-r-00000
```

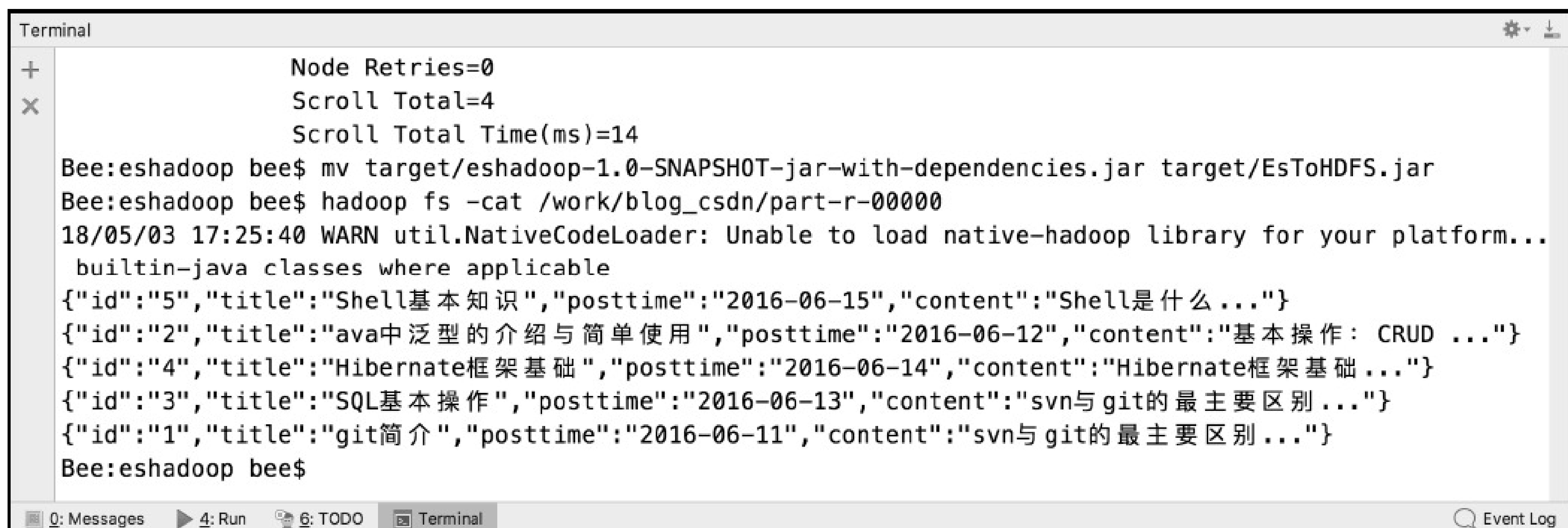


图 11-11　读取 Elasticsearch 数据到 HDFS

11.4.2　查询 Elasticsearch 写入 HDFS

可以传入查询条件对 Elasticsearch 中的文档进行搜索，再把查询结果写入 HDFS。这里查询 title 中含有关键词 git 的文档，代码如下：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io .Text;
import org.apache.hadoop.io .Writable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.elasticsearch.hadoop.mr .EsInputFormat;
import java.io .IOException;
public class EsQueryToHDFS {
   public static class MyMapper extends Mapper<Writable, Writable,
Text, Text> {
      @Override
      protected void map(Writable key, Writable value, Context
      context) throws IOException, InterruptedException {
         context.write(new Text(key.toString()),new
            Text(value.toString()));
      }
   }
   public static void main(String[] args) throws Exception {
      Configuration configuration = new Configuration();
     String[] otherArgs = new GenericOptionsParser(configuration,
                              args).getRemainingArgs();
      configuration.set("es.nodes", "127.0.0.1");
      configuration.set("es.port", "9200");
      configuration.set("es.resource", "blog/csdn");
      configuration.set("es.output.json", "true");
      configuration.set("es.query","?q=title:git");
      Job job = Job.getInstance(configuration, "query es to HDFS");
      job.setMapperClass(MyMapper.class);
      job.setNumReduceTasks(1);
      job.setMapOutputKeyClass(Text.class);
      job.setMapOutputValueClass(Text.class);
      job.setInputFormatClass(EsInputFormat.class);
      FileOutputFormat.setOutputPath(job, new Path(otherArgs[0]));
```

```
        job.waitForCompletion(true);
    }
}
```

打包后执行作业：

```
mv target/eshadoop-1.0-SNAPSHOT-jar-with-dependencies.jar
target/EsQueryToHDFS.jar
hadoop jar target/EsQueryToHDFS.jar /work/EsQueryToHDFS
```

作业执行完成以后，会在 HDFS 的/work 目录下新增一个 EsQueryToHDFS 目录，使用 cat 命令查看数据，命令如下，执行结果如图 11-12 所示。

```
hadoop fs -cat /work/EsQueryToHDFS/part-r-00000
```

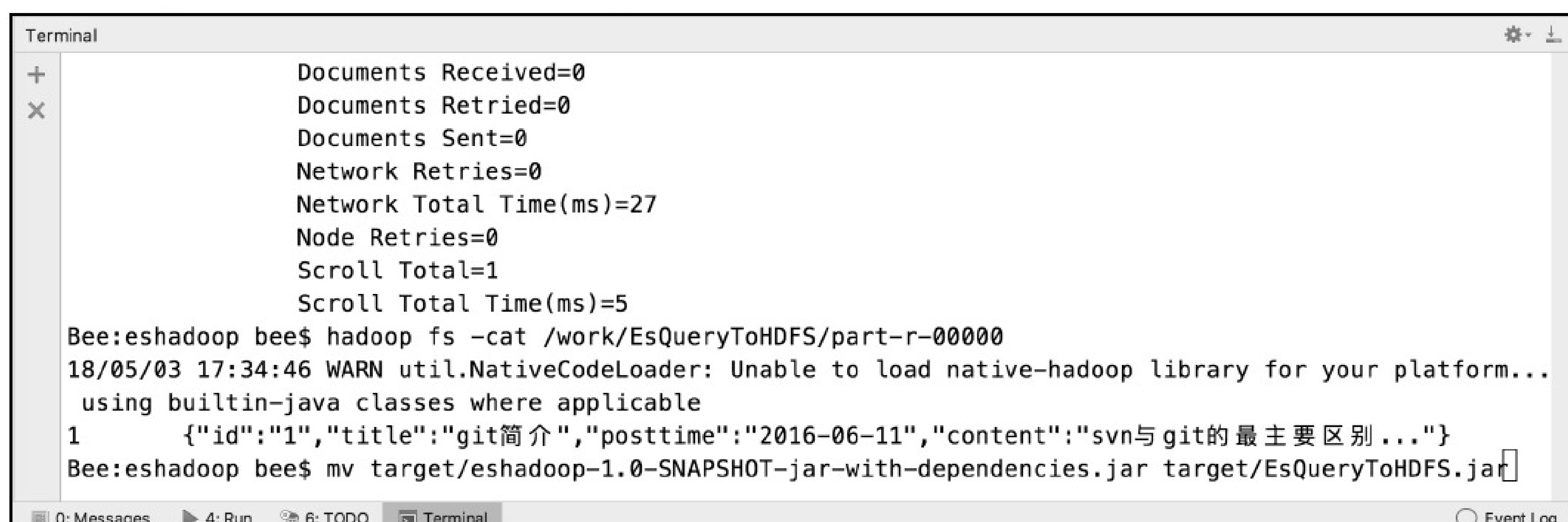


图 11-12　查询 Elasticsearch 数据到 HDFS

11.5　本章小结

本章介绍了连接 Elasticsearch 与 Hadoop 的桥梁 ES-Hadoop，搭建了 Hadoop 伪分布式环境，给出了导入 HDFS 上的数据到 Elasticsearch 以及查询 Elasticsearch 上的数据到 HDFS。通过本章的学习，应该能够掌握 Elasticsearch 与 Hadoop 交互的方法。

参考文献

[1] [美]Christopher D.Manning, [美]Prabhakar Raghavan, [德]Hinrich, Schutze 著; 王斌译. 信息检索导论[M]. 北京：人民邮电出版社, 2010.

[2] 张俊林, 著. 这就是搜索引擎：核心技术详解[M]. 北京：电子工业出版社，2012.

[3] [美]Bing Liu 著;俞勇 译. Web 数据挖掘（第 2 版）[M]. 北京：清华大学出版社，2013.

[4] [波兰]Rafał Kuć, [波兰]Marek Rogoziński 著; 时金桥等译. ElasticSearch 可扩展的开源弹性搜索解决方案[M]. 北京：电子工业出版社，2015.

[5] http://lucene.apache.org/core/documentation.html

[6] https://www.elastic.co/guide/index.html